

The Design and Implementation of a Web Application for Visualizing Chemical Structures and Information Using Test Driven Development

by

©Dayo J. Osunrinde

A Thesis submitted to the School of Graduate Studies in partial fulfillment of the
requirements for the degree of

Master of Science

Department of Scientific Computing

Memorial University of Newfoundland

June 2017

St. John's

Newfoundland

Abstract

This research examined the use of Test-Driven Development (TDD) for the creation of a web application to visualize chemical structures. To date, TDD has mainly been applied to develop traditional software systems. The study however described the novel application of TDD to the development of a high-quality scientific web application using Django, Python 3, Selenium, HTML5, CSS3, JavaScript, JSmol and the RESTful API. With TDD, automated tests are written first. These tests drive the design of the software towards an extensible application that easily accommodate changes and updates. Presented in this thesis is the test driven design, analysis and implementation of the visualization application as well as the tests and verification results. The study concluded that the application of TDD to scientific web application development can lead to a better design, high-quality production code, and easier integration of changes and new requirements. However, the research made no claim that TDD is a magic wand that solves all software problems. Rather, the approach has the prospect of ensuring high-quality web applications as it can be a bit tedious.

Acknowledgements

First and foremost, I wish to express my sincere gratitude to Professor Raymond Poirier a prominent scholar who would never mind to see budding scholars fly. I would also like to thank my co-supervisor Professor Jason Pearson at the University of Prince Edward Island. It is a great pleasure to thank Dr Oliver Stueker, an ACENET consultant and a prominent member of the Retrieivium research group for his continuous support in learning python programming from the scratch and his immense help during my graduate program. I should thank Csongor Matyas, a PhD candidate of the Theory group for many discussions on software development, programming and beyond. I express my sincere appreciation to every member of the Retrieivium group and Theory Lab. Finally, I am grateful to my parents and siblings for always being there. Thanks to everyone I am very grateful.

Table of Contents

Abstract	ii
Acknowledgments	iii
List of Figures	xi
List of Abbreviations and Symbols	xii
1 Introduction	1
1.1 Overview of Project	1
1.2 Goals of the project	3
1.3 Thesis Outline	4
2 Test Driven Development	5
2.1 Introduction	5
2.2 TDD Definition	5
2.3 Simple TDD Example	6
2.4 Test Driven Development Concept	10
2.5 Suggested TDD Efficacy in Context	14
2.5.1 TDD in Academia	14
2.5.2 TDD in Industry	17

2.6	TDD Benefits	23
2.6.1	Higher Production Code Quality	23
2.6.2	Enhanced Application Quality	24
2.6.3	Enhanced Developer Productivity	25
2.7	The Disadvantages of TDD	26
2.8	The Controversies	26
3	Web Application Development	28
3.1	Introduction	28
3.2	Overview	28
3.3	Web Programming	29
3.3.1	Client-Side Programming	29
3.3.2	Server-Side Programming	30
3.4	Application Development Environment	30
3.5	Server-Side Tools	30
3.6	Django Framework	31
3.6.1	Django Overall Design Philosophies	31
3.6.2	How Django Works	32
3.7	Python 3.5.1	33
3.8	Selenium 3.0	34
3.9	Django TestCase	35
3.10	REST API 3.5.4	35
3.10.1	HTTP Methods	36
3.11	Front-End Tools	37
3.12	HTML5	37
3.13	JavaScript 3.10.2	38
3.14	JSmol 3.10.3	39

3.14.1	Main Features of JSmol	39
3.14.2	JSmol Initialization	40
3.14.3	Setting Parameters	41
3.14.4	CSS 3.10.4	42
3.15	Integrated Development Environment	43
3.15.1	PyCharm	43
3.16	Version Control System	44
4	Application Specifications	45
4.1	Introduction	45
4.2	Chemical Markup Language	45
4.3	CML Visualization Web Application	46
4.4	Primary Requirements	46
4.5	Use Cases	49
4.6	Milestones	50
4.7	Design Visualization	52
4.8	Secondary Requirements	53
5	Implementation With TDD	55
5.1	Introduction	55
5.2	Application Overview and Design	55
5.3	Project Structure	57
5.4	Explanation of Application Structure	58
5.5	Setting Up Application Functional Tests	60
5.5.1	Django TestCase	60
5.5.2	Selenium WebDriver	61
5.5.3	Opening Page with WebDriver	62

5.6	Locating Elements	64
5.6.1	Locating by XPATH	65
5.6.2	StaticLiveServerTestCase	65
5.6.3	Set Up and Tear Down	66
5.7	Implementing List View	68
5.7.1	List View	73
5.8	Implementing Upload View	74
5.8.1	Upload View	76
5.9	Implementing Item View	77
5.9.1	Item View	85
5.10	Implementing Search Functionality	99
5.11	Secondary Requirement Implementation	100
5.12	Testing Milestones	102
5.13	Implementing Database	103
5.14	Data Extraction	104
5.15	Implementing Beauty	109
6	Tests and Verifications	111
6.1	Introduction	111
6.2	Tests Outline	111
6.3	Functional Testing	112
6.3.1	Form Validation	113
6.3.2	Information Extraction Testing	113
6.3.3	Visualization	113
6.4	Compatibility Test	113
6.5	Regression Testing	114

7	Conclusions	115
7.1	Future Work	116
	Bibliography	116
A	Email Permission	124

List of Figures

2.1	Simple TDD Example Project Directory Structure	6
2.2	Simple TDD Process.	10
2.3	Overall TDD Process. Adapted from "Test-Driven Development With Python" by Harry Percival, p.47. Copyright 2014 by the O'Reilly Media Inc. Adapted with permission from Harry Percival.	11
2.4	TDD Process with Functional and Unit Tests. Adapted from "Test-Driven Development With Python" by Harry Percival, p.48. Copyright 2014 by the O'Reilly Media Inc. Adapted with permission from Harry Percival.	12
4.1	An informal drawing of the application with Single Molecule Item View.	52
4.2	An informal drawing of the application with Multiple Molecule Item View.	53
5.1	Overall application architecture	56
5.2	The Project directory structure containing the visualization application.	57
5.3	Template inheritance.	59
5.4	Importing modules and setting variables.	61
5.5	A code fragment that demonstrate the opening of page with the class <code>TestWebDriver</code>	63

5.6	Setup and teardown browser	66
5.7	The setting up of Django admin log in test	67
5.8	The test written for listing view.	69
5.9	The implementation of listing view in <code>views.py</code>	70
5.10	The URL configuration of our application.	72
5.11	Listing view HTML code fragment.	72
5.12	The display of listing view showing uploaded CML test files and their corresponding information.	73
5.13	The test written for upload view.	74
5.14	An helper function for upload view.	75
5.15	The implementation of upload view.	75
5.16	Upload view HTML code fragment.	76
5.17	The display of upload view with the browse and upload button. . . .	76
5.18	The test written for Item view.	77
5.19	The implementation of item view.	79
5.20	Item view HTML code fragment.	80
5.21	The visualization of an uploaded molecule in balls and stick model. .	85
5.22	The visualization of an uploaded molecule in wireframe model.	86
5.23	The visualization of an uploaded molecule in spacefill model	87
5.24	The visualization of dipole arrow.	88
5.25	The labeling of partial charges and dipole arrow.	89
5.26	The visualization of Van-der waal surface and dipole arrow.	90
5.27	The visualization of solvent accessible surface.	91
5.28	The visualization of transparent surface.	92
5.29	The mapping of electrostatic potential.	93
5.30	The visualization of multiple molecule item view electrostatic potential.	94

5.31	The visualization of multiple molecule item view solvent accessible surface.	95
5.32	The visualization of multiple molecule item view partial charge. . . .	96
5.33	The visualization of multiple molecule item view ball and stick. . . .	97
5.34	The visualization of multiple molecule item view (3 Molecules) ESP. . .	98
5.35	The implementation of search functionality.	99
5.36	The test written for the secondary requirement.	100
5.37	The test written to ensure total coverage of milestone.	102
5.38	The test written for the database.	103
5.39	The application database (Models).	104
5.40	The implementation of extraction.	105
5.41	The application CSS style sheet.	110

List of Abbreviations and Symbols

TDD	Test Driven Development
CML	Chemical Markup Language
HTML	Hyper Text Markup Language
XML	Xtensible Markup Language
XP	Extreme Programming
MVC	Model, View, Controller
MVT	Model, View, Template
AUD	Application Under Development
REST	Representational State Transfer
WAF	Web Application Framework
API	Application Programming Interface
DTL	Django Template Language
VCS	Version Control System
VDW	Van-der-Waal Surface
SAS	Solvent Accessible Surface
ESP	Electrostatic Potential

Chapter 1

Introduction

1.1 Overview of Project

The evolution of web applications has accelerated and extended into scientific research. The use of web applications to aid scientific research is getting more substantial over the years. In recent times, the portability of web application has made it more preferable to traditional software as computational scientists present their applications through a web interface. In essence, this implies that users can upload input data to an application on a web page, and then click on some buttons that has been programmed to carry out some computations, and display the results as appropriate. In a situation where there are collaborators spread across different geographical locations, then the web interface can be deployed to a server and made accessible to all and sundry.

With the advent of the three tier web application framework and high level programming, scientist have been able create programs designed to perform specific tasks, applicable to their own research in a simple and straightforward manner. This process of program or software development in academia usually involves different programmers over many years, who are mostly students. Over the years, these programs are

extended to accommodate new requirements as area of research dictates. However, the problem with this kind of arrangement is that there is no apparent plan, or standards guiding the process unlike code written in industry. Since most of this code are often written by students, programming best practices are mostly ignored. A good number of project code found in academia are written in this manner. Consequently, most of these codes are ridden with bugs, defects, and errors. The bad code design makes it difficult to extend, as reading and maintaining the source code becomes next to impossible. Hence, the low-quality applications developed in academia. For this reason, there arises a need for a disciplined programming scheme that could raise the standard of applications produced in academia.

Over the years, new methodologies have emerged in software development in a bid to improve development process and products. In 2001, Beck [4] published an article about a new approach of development called Test-Driven Development (TDD). TDD is a test-centered methodology where automated tests drive the design of a software. With TDD, the code is tested before it is written. Till date, TDD has mainly been applied to the development of traditional software systems, and not to the design of any scientific web application. This thesis therefore introduces a new test-driven approach to develop an interactive 3D chemical structure visualization web software with focus on maintainability and high-quality. The ideas in this thesis is however applicable to any kind of web application and software as well.

1.2 Goals of the project

In this thesis, a TDD approach to the creation of web application using Django [10], Python 3 [61], HTML5 [52], CSS3 [58], JavaScript [20], JSmol[60], and the RESTful API [54] is presented. In particular, TDD is examined and applied as the overall programming scheme to develop a high-quality requirements visualization application for Chemical Markup Language [33] (CML) data. CML data have the .cml extension, and are unified with XML-based encoding requirements. These requirements are the implemented guidelines for the storage and the sharing of content related to chemical elements, compounds, analytical work done to experiments using certain chemicals, and scientific or mathematical equations used in chemistry. For instance, the data of these CML files may include descriptions of certain molecular structures entered by the authors of those CML files [48].

The TDD approach is as general as possible and can be applied to any web application with special focus on quality. Chapter 5 gives a detailed description of the application’s specifications. Regardless, a summary of the desired features of the visualization application are as follows:

- The application is expected to be able to upload CML files.
- It should have the ability to extract specified parameters from the uploaded CML files.
- It should have the ability to store the extracted CML files.
- It should have the ability to visualize molecular structures and other properties in different models from the stored CML files.
- It should be highly maintainable and of high-quality.

- It should be scalable and extensible.

The language of implementation will be Python 3 [61], as it uses a neat and simple syntax that makes the written programs readable. Also, Python comes with series of large standard libraries and useful modules that supports many common programming tasks such as data extraction, connecting to web servers, searching text with regular expressions, reading and modifying files. The most popular web framework built on Python called Django will be our web application framework, while the front-end and the back-end will be built using HTML5, JavaScript, JSmol, CSS3 and Python 3 as appropriate (see Chapter 4 for more details).

1.3 Thesis Outline

The outline of this thesis is as follows. Chapter 2 examines test-driven development. It gives detailed description into the process and the supposed benefits in context. Chapter 3 gives a detailed description of web application development, as well as the application development environment and the various tools used in the development of our application. Chapter 4 describes the requirements, project milestones, and specifications in terms of functionality of the application under development (AUD). Chapter 5 provides a description of the TDD implementation of our CML visualization application and results. Chapter 6 gives details about the testing and verification. Chapter 7 provides details about the conclusion and future work.

Chapter 2

Test Driven Development

2.1 Introduction

In this Chapter, a review of context that covers the prior work done regarding TDD in academia and industry, as well as the advantages, disadvantages in TDD-related literature is presented.

2.2 TDD Definition

TDD is a fairly new programming method in which testing, coding and refactoring activities all go in small iterative steps [2, 4]. TDD was initiated by Beck, the originator of extreme programming (XP) [4]. The fundamental thought behind the creation of TDD was to achieve fast, clean code that works. Beck described TDD as a novel approach to software development, where the programmer must first write a test that fails before writing a single line of production code [4]. TDD originated with the birth of XP and thus became a key practice of XP [3]. The main goal of TDD is specification and not validation [37].

Otherwise stated, it is one way to think through the given requirements or design before writing any single line of production code (implying that TDD is both an important agile requirement and agile design technique). Another view is that TDD is a programming technique with the overall goal of writing clean code that works as stated by Ron Jeffries [31].

In essence you follow three simple steps repeatedly [21]:

1. Write a test for the next piece of functionality you want to add.
2. Write the functional code until the test passes.
3. Refactor both new and old code to make it well structured.

2.3 Simple TDD Example

In this trivial example, we want to write a Python [61] function that adds two numbers and returns the sum as the output. The first step is writing a failing test before writing any single line of production code. A new project called **sample** was created, with two separate directories named **Additionapp** and **test** in the root project folder. The Figure below shows the directory structure in Pycharm [47] development environment.

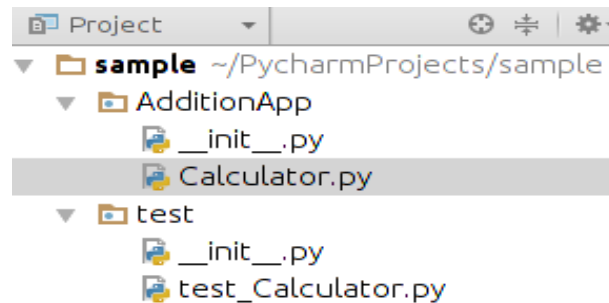


Figure 2.1: Simple TDD Example Project Directory Structure

According to TDD, everything starts with test. So we start by writing a test as seen in the following lines:

```
1. import unittest
2. class TddSimpleExample(unittest.TestCase):
3.     def test_add_two_integers(self):
4.         calc = Calculator()
5.         result = calc.add(50,50)
6.         self.assertEqual(100, result)

7. if __name__ == '__main__':
8.     unittest.main()
```

Here is the break down of each line in the code above.

- In the first line, the `unittest` Python standard module for unit testing was imported.
- In the second line, a test class that was made a subclass of `unittest` module was created.
- In line three, a function was defined, and assigned to do the calculation.
- In line four and five, a function call was made to the `Calculator` function. The return value was assigned to the newly created variable `result`.
- In line 6, a method that asserts if the expected output is what the function returns as output was called from the `unittest` module.

We then run our application by running the test, hoping to get our failing test.

```

$ python3 test_calculator.py

E
=====
ERROR: test_calc_add_two_integers (test.test_add_two_integers.TddSimpleExample)
-----

Traceback (most recent call last):
File "/Users/user/PycharmProjects/sample/test/test_Calculator.py",
line 6,in test_add_two_integers
calc = Calculator()
NameError: global name 'Calculator' is not defined
-----

Ran 1 test in 0.001s
FAILED (errors=1)

```

As expected, what the error is simply telling us is that we are trying to import what we have not created. So in our `Calculator.py` file, we write the following lines of code, after which we will have to modify the test by importing the `Calculator` from the `AdditionApp` directory. (See line 2 of the `test_Calculator` file as shown below):

```

class Calculator(self):
    def add( x, y):
        pass

1. import unittest
2. from AdditionApp.Calculator import Calculator
3. class TddSimpleExample (unittest.TestCase):
4.     def test_add_two_integers(self):

```

```

5.         calc = Calculator()
6.         result = calc.add(2,2)
7.         self.assertEqual(4, result)

8. if __name__ == '__main__':
10.     unittest.main()

```

Running the test again after the above update, we have the error below.

```

$ python3 test_Calculator.py
E
=====
FAIL: test_calc_add_two_integers (test.test_Calculator.TddSimpleExample)
-----
Traceback (most recent call last):
File "/Users/user/PycharmProjects/tdd_in_python/test/test_calculator.py"
line 9, in test_calc_add_two_integers
self.assertEqual(4, result)
AssertionError: 4 != None
-----

Ran 1 test in 0.001s
FAILED (failures=1)

```

Obviously, our method is returning the wrong value, as it doesn't do anything at the moment. The next step is then fixing the method and ensuring that our test passes.

```
class Calculator(object):  
    def add(self, x, y):  
        return x+y
```

```
$ python3 test_Calculator.py
```

```
.
```

```
-----  
Ran 1 test in 0.000s
```

```
OK
```

What we have just done is a simple example of a TDD approach. We have started with a test, written minimum code to get our test to pass one step at a time. Assuming this was a large project, we may need to refactor as we go along adding functionality one step at a time.

2.4 Test Driven Development Concept

TDD involves the simple process as shown in Figure 2.2.

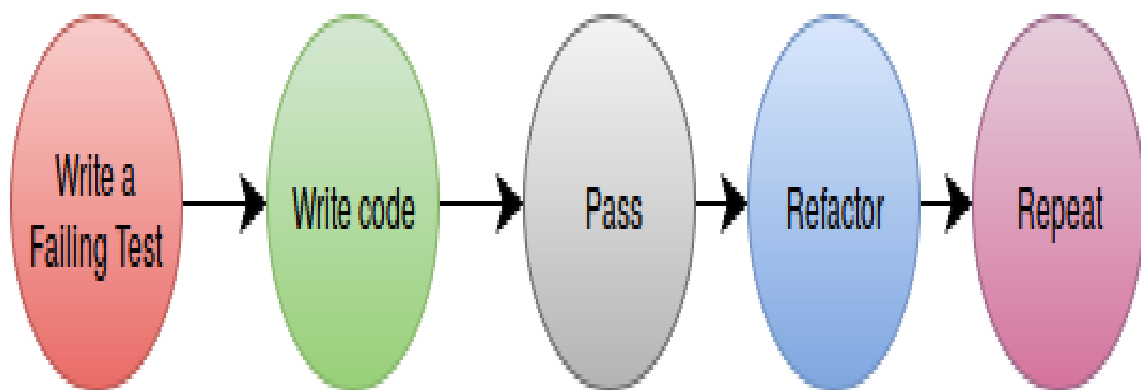


Figure 2.2: Simple TDD Process.

Similarly, Percival in 2014 [46] suggested a workflow in his book Test-Driven Development with Python where he defined the overall TDD process as given in Figure 2.3.

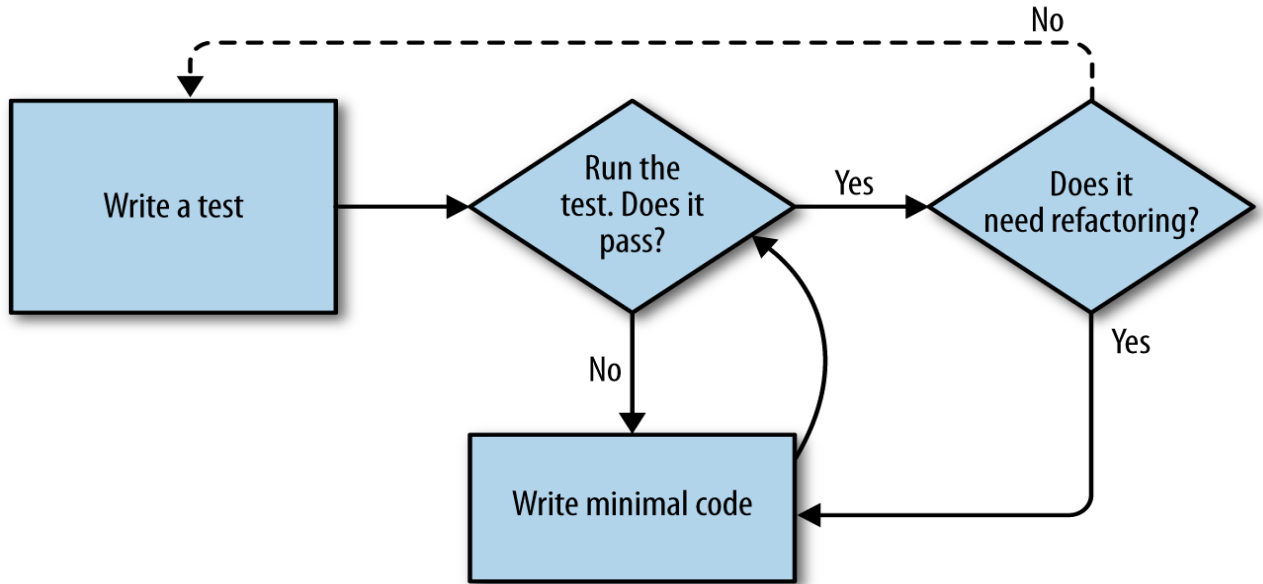


Figure 2.3: Overall TDD Process. Adapted from "Test-Driven Development With Python" by Harry Percival, p.47. Copyright 2014 by the O'Reilly Media Inc. Adapted with permission from Harry Percival.

1. Write a test.
2. Run the test and see it fails.
3. Write some minimal code to get the test to pass.
4. Rerun the test and repeat until it passes.
5. Optionally, we might refactor our code, using our tests to make sure we don't break anything.

Harry also describes the main aspects of the TDD process in practice as [46]:

- Functional tests

- Unit tests
- The unit-test/code cycle
- Refactoring

In this context, where we have both functional test and unit test together, we can liken functional test to being a high-level view of the cycle, where writing the code to get the functional tests to pass actually involves using another, smaller TDD cycle which uses unit tests. This is illustrated in the unit test code cycle in Figure 2.4 [46]:

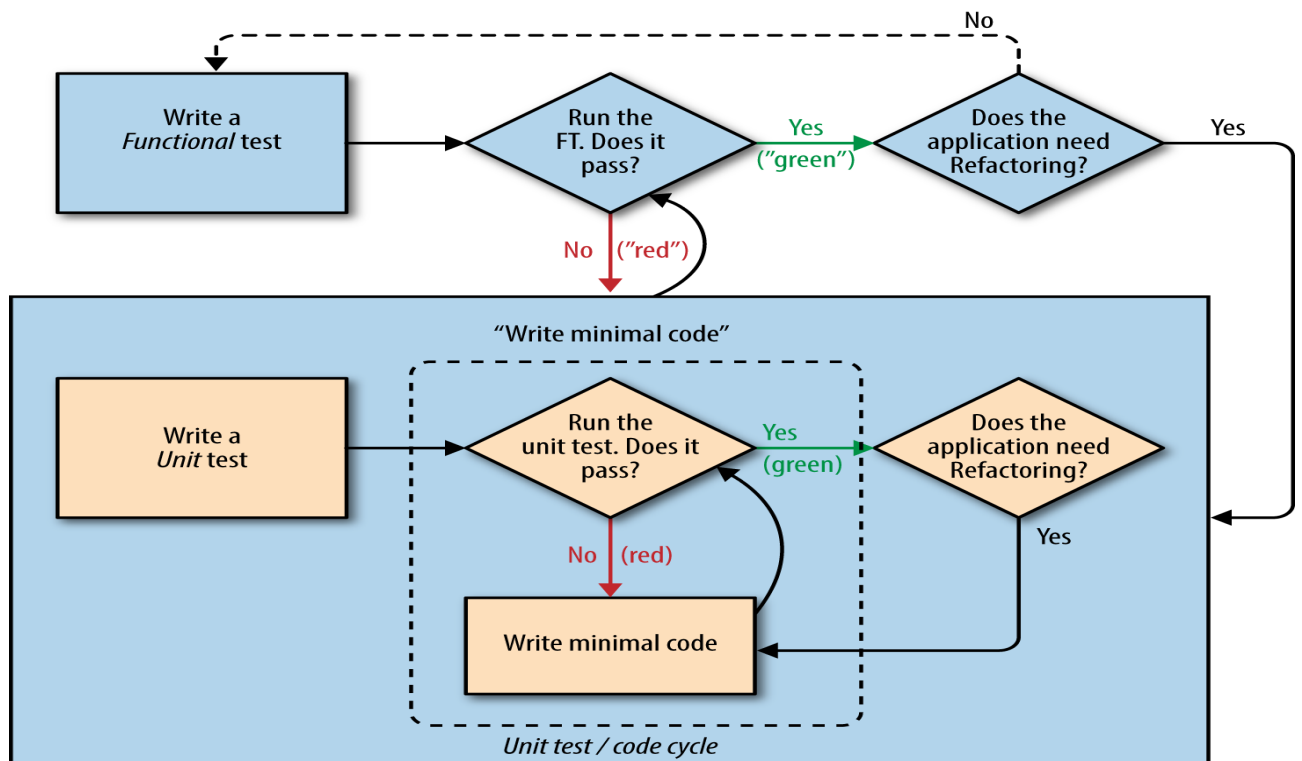


Figure 2.4: TDD Process with Functional and Unit Tests. Adapted from "Test-Driven Development With Python" by Harry Percival, p.48. Copyright 2014 by the O'Reilly Media Inc. Adapted with permission from Harry Percival.

In the simplest terms, TDD inclusive of both functional and unit tests means write a functional test, see it failing, then divide the problem into parts, write a unit test for each part, write code for each part, see unit tests pass and then your functional test should pass. The functional tests are the ultimate judge of whether your application works or not, while the unit tests are a tool to help you along the way [46]. This leads us to the definition of the core concepts in TDD:

1. **Functional Testing**

The main objective of functional testing is to test each discrete component of an application from the point of view of the developer. It seeks to ensure that all the given specifications are implemented perfectly and they work as expected. These tests allow us to see how an application works from the perspective of the user. Moreover, functional tests capture how the user might work with a series of requirements and how the application should respond to them. During the development, Selenium [50] was used to test the functionality of our application.

2. **Unit Testing**

Unit testing allow the testing of a very small piece of code at the application programming interface (API) level. Running unit tests is quite easy as it does not require a full production environment to run. With Pycharms [47] it becomes easier at the click of a button. For our development, we have used Django `TestCase` which makes use of the Python `unittest` module for unit testing.

3. **Refactoring**

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, and yet improves its internal structure. Refactoring improve the design of a code after it has been written [22]. It is the last step to completes each of the TDD's iterative steps. In this

context, it utilizes a functional testing to ensure that we have preserved the behavior of our application while cleaning up bad code.

2.5 Suggested TDD Efficacy in Context

This section describes prior research work done on TDD. More specifically, literature review that covers experimental, comparative, and empirical TDD-related research published at conferences, and in journals, is presented. It also covers a brief overview of the project undertaken, challenges, observations, and the results. These are structured into two sub-sections as corresponding to work done within academia and industry.

2.5.1 TDD in Academia

The most popular reference to the TDD approach was a study facilitated in 2002 by Muller and Hagner [40] at the University of Karlsruhe. The purpose of this experiment was to evaluate [40]:

- The programming efficiency of TDD (how fast a person obtains a solution).
- The reliability of the code (how many failures can be found, measured as a part of the passed assertions associated with all feasible executable assertions in the test).
- Program understanding (measured as proper calls of existing methods).

The study consisted of 19 students divided into two groups. While 10 of the students were mandated to use test-first (TFG group), the other group was mandated to use the traditional test-last approach and they were called the control group (CG). On average, the students had 8 years programming experience.

The experiment was divided in two phases [40]:

- Implementation Phase (IP)

In this phase, the students developed their program in readiness for acceptance test.

- Acceptance-test Phase (AP)

In this phase, students in both groups fixed faults that caused the acceptance test to fail.

At the end of the first phase (IP), test results show that the reliability of the programs from the TFG group, were significantly higher than reliability of programs from the CG group. Conclusively, the following important points were made from the experiment [40]:

- Five programs from the TFG have over 96% reliability.
- Test-first delivers better reliability on overall.
- Test-first programmers reuse current methods efficiently and swiftly. This is caused by on-going testing approach and while fixing the fault, developer learns about existing code.
- If the developer switches from traditional development to a test-first approach it does not imply that he can find the solution more quickly.

Mustafa, in his thesis [44] proposed TDD as a better alternative to an established software development practice in a large firm that develops embedded software. He recommended a methodology that integrates TDD into a classical development cycle without necessitating an outright transition to agile methodology of software development. His case study was an embedded software development project. He concluded within the scope of his project, that TDD is a viable approach.

In 2005, Erdogmus et al. [19] facilitated a controlled experiment for evaluating an important aspect of TDD. The experiment was conducted with undergraduate students divided into two groups. The experiment group practiced TDD, while the control group applied the conventional development technique which involved writing test after implementation. Both groups followed an incremental process of adding new feature one step at a time and performing a regression test simultaneously after implementing each feature.

The assessment focused on the test-first characteristic of TDD, where programmers implement a small piece of functionality by writing a unit test before writing the corresponding production code. On the grounds that test-first is basically a step-by-step method, they designed the test to fit in an incremental development context.

The final conclusions were [19]:

1. Test-first appears to improve productivity. From the results, it was concluded that advancing development one test at a time and writing tests before implementation encourages better decomposition.
2. Test-first improves the mastery of the underlying specifications and minimizes the purview of the tasks to be performed.

3. The small confines of the tests and the fast reversal made possible by continual regression testing all at once reduced debugging and rework effort.

In summary, the effectiveness of the test-first approach would possibly depend entirely on its capability to encourage programmers to back up their code with test assets.

Gupta and Jalote [24], evaluated the impact of TDD on various program development activities like designing, coding, and testing, through a controlled experiment where they compared it with the conventional way of developing the code. In a single-factor block design, two groups of students developed two moderately-sized programs following the two development-styles under study [24]. The results suggested that:

- TDD helps in reducing overall development effort and improving developer's productivity.
- The code quality appears to be affected by the actual testing efforts implemented during a development-style.

2.5.2 TDD in Industry

From the industrial viewpoint, there are several studies that has evaluated the efficacy of TDD. In a bid to evaluate TDD at IBM in 2003, a software development group at IBM Retail Store Solutions in North Carolina built a non-trivial software system based on a stable standard specifications using TDD in comparison with the conventional test-after method [38]. The main aim of the research was to examine [38]:

1. TDD practice within the context of more robust design.
2. Smoother code integration.

From this study they concluded that [38]:

1. TDD reduced defect rate because they observed a dramatic 50% improvement in the defect rate of the built system.
2. TDD allowed the creation of automated unit test cases that are reusable, extendable, and that will continue to improve in quality over the lifetime of the software.
3. TDD practice helped in producing a product that could change more easily and incorporates late changes.
4. The test suite can also serve as the basis for quality checks, and quality contract between team members.
5. TDD create a significant suite of reusable and extendable regression test case asset that continuously improves quality over software lifetime.

In 2003, another empirical and comparative study was facilitated by George and Williams [23]. The purpose of this research project was to evaluate the following hypotheses:

1. That TDD practice will yield superior external code quality when compared with code developed with a more traditional waterfall-like practice. External code quality will be evaluated based on the number of passed functional (black-box) test cases.
2. That programmers who practice TDD will develop code faster than programmers who develop code with a waterfall-like practice. Programmers speed will be calculated by the time to complete (in hours) a specified program.

To this end, three TDD experiment trials were executed with 24 professional programmers who had experience levels with TDD from beginner to expert at three companies (John Deere, RoleModel Software, and Ericsson). In each of the experimental trials, the programmers were randomly assigned to one of two groups: TDD or Control. All programmers used the pair programming practice. Developers were divided in the group of six pairs developers that used the TDD approach and control group of six pairs that used conventional test-last approach. Each pair was asked to develop a bowling game application with given specifications. The control group pairs used a conventional design-develop-test (similar to waterfall) approach. Participants were asked to turn in their programs upon completing the activities as outlined. Then, their projects were assessed quantitatively and qualitatively. A qualitative survey that centers around the questions listed below was conducted during the research and the response was analyzed [23]:

- How productive is the practice for programmers?
- How effective is the practice?
- How difficult is the approach to adopt?

The following is a summary of the answers:

- As regards the productivity question, 87.5% of the developers felt that the TDD approach made them have a better understanding of the requirements. 95.8% felt that TDD reduces debugging effort, while 78% of the developers thought that TDD improves overall productivity of the programmer.
- In response to the effectiveness question, 71% thought the approach was noticeably effective, 79% felt that TDD promotes simpler design, 92% of the developers believed that TDD yields higher-quality code.

- In response to the difficulty question, 56% of the professional developers thought that getting into the TDD mindset was difficult. 23% indicated that the absence of upfront design phase in TDD was an obstruction. 40% of the developers thought that the approach faces difficulty in adoption.

Finally, six conclusive points were made by the authors [23]:

1. TDD approach seems to yield superior external code quality.
2. TDD development took 16% more time for development.
3. 80% of developers believed that TDD was an effective approach, while 78% believed that the approach improved programmer's productivity.
4. TDD facilitates simpler design and lack of up-front design is not a hindrance.
5. For some, transition to the TDD mindset is difficult.

Bhat and Nachiappan [5], evaluated the efficacy of TDD at Microsoft Corporation. They performed two case studies using TDD in two Microsoft divisions: Windows and MSN, using a project of the same level of complexity. The main focus in this project was to:

1. Compare the differences in software quality between the TDD and non-TDD methodology.
2. Compare the overall development time when using the TDD and non-TDD methodology.

In both cases, they measured the various contexts, products and outcome measures to compare and evaluate the efficacy of TDD. They observed a significant increase in code quality (greater than a factor of two) when using TDD, compared to similar projects developed in a non-TDD manner. The projects also took at least 15% extra upfront time for writing the tests. Additionally, the unit tests served as auto documentation for the code. Also, libraries/APIs were used for code maintenance. Final conclusions in this paper was that TDD produced high code quality.

Wasmus and Gross [56], of the Software engineering research group at the Delfts University of Technology also evaluated TDD in industry. The following reasons constituted the purpose of their research:

1. They wanted to see if TDD incorporates requirements changes easily.
2. They wanted to see if TDD leads to superior technical solution in software.
3. They wanted to find out if TDD results in better and cleaner code and motivates all stakeholders.

They presented a development project carried out in a company to ascertain those claims. The evaluation was part of an industry-scale application development project at EPCOS Inc. The application was a forecasting system that was meant as a pilot study to decide whether TDD could be introduced as a new development paradigm across software development units at EPCOS. The final conclusions were:

- The biggest advantage of TDD is that a product of high-quality can be developed by maintaining flexibility.
- TDD produces maintainable high-quality source code.

Ambler [1], of IBM extended TDD to database development which he called test-driven database development. From his garnered experience and in conclusion, he claimed that test-driven database development is important for several reasons highlighted below:

1. First, all of TDD's benefits extend to test-driven database development. With test-driven database development, developers can take small, safe steps.
2. He stated further that refactoring let developers maintain high-quality design throughout the life cycle.
3. Regression testing provides the opportunity to detect defects earlier in the life cycle because test-driven database development gives an executable system specification, and motivates developers to keep it up-to-date.
4. With TDD, database development efforts effectively dovetail into the overall application development effort.

In a follow-up experiment in 2008, Nagappan [45] conducted another study that involved three development teams at Microsoft and one development team at IBM that have adopted TDD. The results of indicate that:

1. The four products recorded a decrease in pre-release defect density. While IBM had a 40% decrease, Microsoft had between 60% to 90%. These are relative to similar projects where TDD practice was not implemented.
2. From a subjective point of view, the teams recorded an increase between the range of 15–35% in initial development time after the adoption of TDD.

2.6 TDD Benefits

The often mentioned advantages from the literature, are higher production code quality [5, 23, 29, 34, 56], enhanced application quality [1, 5, 18, 23, 27, 38, 40, 56], and enhanced developer productivity [19, 24, 28]. These will be explicitly discussed in the following subsections.

2.6.1 Higher Production Code Quality

A good deal of articles, blogs and empirical studies in the literature claimed that TDD improves code quality. In a bid to ascertain or question this claim, Jansen and Saiedian [29], conducted three quasi-controlled experiments and one case study in a Fortune 500 company and another two quasi-controlled experiments with university students in undergraduate and graduate software engineering courses. The main focus of these experiments were on internal software quality with respect to design and code characteristics such as:

- Code complexity.
- Size.
- Coupling and cohesion.

The result indicated that TDD programmers wrote software modules that are smaller, less complex and more highly tested modules than modules produced by their test-last counterparts. Similarly, Wasmus et al. [56] (experiment details given in the preceding session) concluded from their experiment that TDD produced maintainable high-quality source code.

Dogsa and Batic [18], conducted a multi-case study investigating the effectiveness of TDD within an industrial environment with respect to code quality, productivity and maintainability. Three comparable medium-sized projects were observed during their development cycle. While two of the three projects were implemented without TDD practice, the third introduced TDD into the development process. Their results indicated that the third project had maintainable higher-quality code in comparison with the first two.

Crispin [34], in her work shared her experience working on different development teams that used TDD and vice versa. She said "I can tell you from first-hand experience that TDD produces code that has orders-of-magnitude fewer unit-level bugs, far fewer functional bugs, and an exponentially higher probability of meeting stakeholder expectations when compared to code produced by conventional programming techniques". Crispin also provided evidence from her experience that code written using TDD is easier to understand.

2.6.2 Enhanced Application Quality

TDD evangelists claim that adherence to this approach can simultaneously improve both quality and productivity [3, 27]. In the same vein, conclusions from the most popular TDD effectiveness reference Muller and Hagner [40], as well as result from Crispin's [34] experience also attested to this claim. They indicated that TDD leads to better requirements understanding and this invariably improves the quality of the software. Similarly, the conclusions of industrial studies [5, 23, 27, 34, 38, 56] have provided strong evidence that TDD facilitates the production of high-quality code, in comparison with projects developed using conventional approach.

2.6.3 Enhanced Developer Productivity

Advocates of TDD claim that it enhances the overall productivity of a development team by reducing total project development effort defined as time spent directly on the project including analysis, design, code, test, fix and review [29]. TDD proponents also asserted that TDD improves overall developers efficiency as a result of tested code written from the beginning of the development. This is believed to save time expended during the regular testing phase [24].

Also in TDD, each unit of functionality according to the user story is written as a functional test. This takes into account how far and how well the process is going, hence a source of motivation for the developer [19]. During TDD, logical bugs and regression bugs are fixed earlier, this reduces the excessive amount of time needed to fix defects in comparison to the test-last approach. All these in turn could increase productivity.

Williams et al [38], asserted that TDD reduces the overall amount of time spent on debugging, rework and bug fixes following post-release failure, with a resulting increase in overall long-term productivity. Cannam suggested that by adopting TDD, developers are more likely to gain a diverse thinking on how to develop logic and algorithms in their code, and subsequently it can give them safety and confidence in their work [6].

Conclusions from empirical studies reviewed are in support of this claim [19, 24, 28]. Results from these experiments, show that TDD programmers were more productive than the test-last programmers since the TDD team spent less effort per line-of-code, and 57% less effort per feature than the test-last team. Also, TDD programmers saved upfront-design efforts. They were more productive in terms of overall development

effort, as there is less rework effort at the testing and fixing stages.

2.7 The Disadvantages of TDD

Some disadvantages of TDD have been identified. Thirumalesh and Nachiappan [5] from their experiment observed that in some cases, the amount of code required by TDD nearly doubled. Atul and Pankaj. [24], also observed from their experiment that TDD could likely decrease code quality. While highlighting some shortcomings of TDD, George and Williams, Thirumalesh and Nagappan, [5, 23] noted that the use of TDD could increase the time for the coding phase. George and Williams [23], also indicated that transitioning to the TDD mindset is difficult, and it might be one factor that increases the efforts put in the coding phase with the use of TDD.

2.8 The Controversies

In defiance of the proven advantages from various studies, TDD has remained a controversial approach among several developers. In 2014, Hanson [25], the author of Ruby on Rails and the founder of Basecamp, gave the opening keynote at Railsconf 2014, in which he challenged the value of TDD. Later, he went on to write two posts on his blog titled "TDD is Dead-Long Live Testing" and "Test-induced design damage". Citing his experience from TDD, he asserted that faith in TDD can lead to completely forgetting about system testing. He also stated that driving design from unit tests is not a good idea, focusing on unit and the unit only does not help in producing a great system and 100 percent coverage is silly. This generated a heated debate among developers all over with some for and some against. However in response to Hanson's assertion, Bob Martin a renown software engineer said in a blog article titled "Monogamous TDD" that "If you aren't doing TDD, or something as effective as TDD, then

you should feel bad.” He further argued that we do TDD for one overriding reason and several less important reasons [36]. The less important reasons are [36]:

1. We spend much less time debugging.
2. The tests act as an accurate and clear documentation at the modest level of the system.
3. Writing tests first encourages decoupling which is generally accepted to be advantageous. These are the debatable supplementary benefits of TDD. However, there is one benefit that that cannot be debated provided that some conditions are met.
4. In the event that you have a test suite that you trust so much that you are willing to deploy the system in light of those tests passing; and if that test suite can be executed in seconds, or minutes, then you can swiftly and effortlessly clean the code without fear of breaking anything.

Fowler [21], hosted a series of recorded hangout conversations between Beck, Hansen and himself, [4, 21, 25] to explore the use of TDD and its impact on software design. This was in a bid to understand each person’s point of view from personal experiences and possibly reach a common ground. Eventually, they all attested to the fact that TDD has values in some contexts.

Chapter 3

Web Application Development

3.1 Introduction

In this Chapter, an overview of web application development, and a detailed description of the tools used for building our visualization application is given.

3.2 Overview

Web development is all about exchange of information between two parties over the HTTP protocol. In Computer Science two systems communicating using the same set of rules is known as protocol. A web application is an application that is invoked with a web browser over the Internet [30]. The procedure and practice of developing web application is called web application development. Conallen loosely defined a web application as a web system (web server, network, HTTP, browser) in which user input (navigation and data input) affects the state of the business [9].

Over the years, web application development has evolved technology-wise and craft-wise through the advent of various web frameworks and advanced tools. At the hub of every web application's work-flow is a request and a response. A request is made through the browser, the server responds to the request by serving the requested page as appropriate, and the browser displays the served response to the user. Everything done is within the request-response model.

3.3 Web Programming

Web programming is defined as the writing, markup and coding involved in web development. It also includes web content, web client and server scripting and network security. Web programming is different from just programming, since it requires interdisciplinary knowledge of the application area, client and server scripting, and database technology [53]. Web programming can be separated into client-side and server-side programming.

3.3.1 Client-Side Programming

The client-side programming, otherwise known as front-end programming is defined as the general name for all programs that runs on the client (browser). It includes a series of languages that the browser understands. In terms of functionality, these languages are used to make interactive and dynamic web pages, send requests to the server and retrieve data from it. The predominant languages used are HTML, CSS and JavaScript.

3.3.2 Server-Side Programming

Server-side programming, otherwise known as back-end programming simply describes the general name for all kinds of programs that run on the server. In terms of functions, server-side programming takes care of processing user input, displaying pages, structuring web applications, and interacting with files. Examples of languages used in server-side programming are PHP, Python, ASP.NET, Java and a host of other everyday programming languages.

3.4 Application Development Environment

An application development environment (ADE) is the hardware, software and/or the computing resources required for building software applications [53]. ADE include the basic hardware infrastructure, such as servers, computers and handheld devices, that will host the application. These are combined with the software engineering resources, such as a programming language's Integrated Development Environment (IDE), and other performance evaluation software utilities.

3.5 Server-Side Tools

The following are the tools used in the project for the back-end or the server side programming. Tools in these categories range from the web framework, server programming language, database, web servers, etc.

- Django framework (1.10)
- Python (Python 3.5.1)
- Selenium (3.0)

- Django TestCase (from Python `unittest` module)

A detailed description of these tools will be given in the following sections.

3.6 Django Framework

Django is a high-level Python web application framework that encourages rapid development and clean, pragmatic design. It was built by experienced developers and it takes care of much of the hassle of web development [10]. Another advantage is that Django is free and open source. Examples of popular sites built on the Django framework are Pinterest, Instagram, Bitbucket, mozilla, the onion, Reddit Gifts and many more.

3.6.1 Django Overall Design Philosophies

Like any substantial design, the developers of Django had some ideas in mind while building the framework. Fortunately, these fundamental ideas align with the goal of creating a maintainable application for this project. The fundamental ideas are [10]:

- **Loose Coupling**

Django framework was built in such a way that every component of the framework is independent from each other. The layers of Django framework does not know about each other. For example, the template system knows nothing about web requests, the database layer knows nothing about data display and the view system does not care which template system a programmer uses. Consequently, Django framework creates maintainable and flexible web applications.

- **Less code**

The more code the more errors. Django framework ensures that the amount of code used in building applications is minimized. This in turn accelerate the development.

- **Better explicitly than implicitly**

Django was designed in a simple way such that it provides ease of use for developers. Therefore, it is easier to make customized changes and tuning to the framework as seem desirable by developers.

- **Rapid development**

The main reason behind the evolution of framework is to make the tedious aspects of web development faster. Django allows for incredibly quick web development.

- **Don't repeat yourself (DRY)**

Django follows the DRY principle that eliminates all forms of redundancy. In Django, each application's functionality is in one place. This does not only reduces the amount of code, but also contributes to the simplicity of the entire application [10].

3.6.2 How Django Works

Django works well for a database-driven website. A database-driven website has the database as the key instrument, and the data for the pages are mainly gained from the database. Django supports four database engines: PostgreSQL, SQLite3, MySQL and Oracle. For the project of this thesis, SQLite3 database was used, because it is already imported in Ubuntu. Django is designed to encourage loose coupling and strict separation between discrete pieces of an application. Django was built in such a way

that each discrete piece of a Django-powered web application has a single key purpose and can be changed separately without having an effect on the other pieces. The Model View Controller (MVC) pattern of software architecture is a way of developing software which separates three fundamental layers of a software application.

Nevertheless, the developers of Django recounted an idea of MVC as when the “view” describes the data that gets presented to the user [13]. In Django, the controller is the framework itself. The controller is described as the machinery that sends a request to the appropriate view, according to the Django URL configuration. Therefore, Django is a “MTV” framework that is, “model”, “view”, and “template” [13].

3.7 Python 3.5.1

Python is a popular, general-purpose, and high-level programming language. It was created by Guido Van Rossum, and first released 1991. The choice of Python for this project centers around code readability, and a syntax which allows the expression of concepts in as few lines of code when compared to other languages such as C++ or Java [35, 39]. Furthermore, Python provides design that encourages the writing of both small and large scale comprehensible programs [32]. Other general notable features of Python are listed below [61]:

- Python uses a neat and simple syntax that makes the programs you write readable.
- Python comes with series of large standard libraries and useful modules that supports many common programming tasks such as data analysis, data cleansing, connecting to web servers, searching text with regular expressions, reading and modifying files.

- Python has an interactive mode that makes it easy to test short snippets of code. It also has a bundled development environment called IDLE.
- Python is extendable by adding new modules implemented in a compiled language such as C or C++.
- Python can also be embedded into an application to provide a programmable interface.
- Python supports several platforms, including Mac OS X, Windows, Linux, and Unix.
- Python is free software in two ways. It doesn't cost anything to download or use Python, or to include it in an application. Python is available under an open-source license [61].

3.8 Selenium 3.0

In this project, we have used Selenium [50] for functionality testing. Selenium is primarily used for automating web applications in order to test the functionality from the user's point of view. The fact that several popular browsers support Selenium as it is also an important technological component in many browser automation tools, APIs and frameworks makes it a perfect choice for the project.

The most recent and biggest change in Selenium as of late has been the incorporation of the WebDriver API. WebDriver is designed in a simpler and more concise programming interface along with addressing some constraints in the Selenium-Remote Control (RC) API [50]. WebDriver is a compact Object Oriented API when compared to Selenium 1.0. It drives the browser much more efficiently and overcomes the

constraints of Selenium 1.x which affected our functional test coverage, like the file upload or download, pop-ups and dialogs barrier. WebDriver overcomes the limitation of Selenium RC [50].

3.9 Django TestCase

Unit testing in Django uses Python's `unittest` module which defines tests using a class-based approach. It is possible to integrate other Python tools and testing frameworks through the API. With Django's test-execution framework and assorted utilities, we simulate requests, inject test data, examine the application's output and for the most part certify that our code is doing what it ought to do [15].

3.10 REST API 3.5.4

REST was first introduced by Roy Fielding in 2000 [54]. It stands for REpresentational State Transfer. REST is a web standards based architecture and uses the HTTP Protocol for data communication. It revolves around resources where every component is a resource and a resource is accessed by a common interface using HTTP standard methods [54]. REST design is made up of two fundamental components namely; a REST Server that gives access to resources and the REST client that access and presents the resources. Every resource is recognized by URIs/ Global IDs. Examples of different representation REST utilizes in representing resources are Text, JSON, and XML.

3.10.1 HTTP Methods

The following HTTP methods are most commonly used in a REST- based architecture [54].

- GET – Gives a read only access to a resource.
- PUT – Used to create a new resource.
- DELETE – Used to remove a resource.
- POST – Used to update an existing resource or create a new resource.
- OPTIONS – Used to get the supported operations on a resource.

Percival [46], described REST as an approach to web design that is usually used to guide the design of web-based APIs. When designing a user-facing site, it is not possible to stick strictly to the REST rules, but they still provide some useful inspiration. REST suggests that we have a URL structure that matches our data structure, in this case lists and list items. Each list can have its own URL [46]:

```
/lists/<list identifier>/
```

To view a list, we use a GET request (a normal browser visit to the page). To create a brand new list, we'll have a special URL that accepts POST requests:

```
/lists/new
```

To add a new item to an existing list, we'll have a separate URL, to which we can send POST requests:

```
/lists/<list identifier>/additem
```

We implemented the RESTful API in our web application.

3.11 Front-End Tools

The following tools were used in the project for the front-end or client-side programming.

- HTML5
- JavaScript was used in the front end for interactive display.
- JSmolScript is a 3D scripting language for molecular visualization.
- CSS3

Each of these tools will be described in the following Sections.

3.12 HTML5

HTML (Hypertext Markup Language) [59] was used for defining the structural design of the application's web pages. However, other features of HTML as highlighted in the documentation are as follows [59]:

- HTML is used in publishing online documents with headings, text, tables, lists, photos, etc.
- HTML is used in retrieving online information via hypertext links, at the click of a button.
- HTML is used in designing forms for conducting transactions with remote services, for use in searching for information, making reservations, ordering products, etc.
- With HTML, spread-sheets, video clips, sound clips, and other applications can be directly included in their documents.

HTML5 is the most recent version of HTML. HTML5 has been designed to deliver almost everything needed to be done online without requiring any plug-ins. Moreover, HTML5 can be used to write web applications that work offline. It handles perfectly high definition video and delivers high-quality graphics [52].

3.13 JavaScript 3.10.2

According to JavaScript's documentation [20], JavaScript is a platform independent, object-oriented scripting language. It is a small and lightweight language that runs inside a host environment such as a web browser. It consists of a standard library of objects, such as Array, Date, and Math, and a core set of language elements such as operators, control structures, and statements. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects. For instance [20]:

- Client-side JavaScript extends the core language by providing objects needed to control a browser and its Document Object Model (DOM). For instance, client-side extensions enable the application under development to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.
- Server-side JavaScript extends the core language by providing objects needed to run JavaScript on a server. For instance, server-side extensions enable an application to communicate with a database, provide continuity of information between the various invocations of an application, or perform file manipulations on a server [20].

3.14 JSmol 3.10.3

JSmol (also Jmol) is a JavaScript framework that enables us to create pages that utilize HTML5, which [60] enables JSmol to display interactive 3D molecular structures in any internet browser that supports HTML5 standards. JSmol is not a distinct program than Jmol, rather it is Jmol compiled into JavaScript instead of Java (using the Java2Script software).

Jmol is a Java applet that was used to view cml molecular data in the application's web page. It read scripts that are contained in Jmol buttons. These scripts are used in the rendering of the molecule to illustrate crucial structural properties in different visualizations. At the center of JSmol is the Jmol JavaScript object, (`window.Jmol`), which comprises a set of JavaScript functions and utilities.

3.14.1 Main Features of JSmol

The following are the main features of JSmol as written in the documentation [60]:

- **Non-Java Options**

Options for Java, HTML5/WebGL, or HTML5-only. Includes a variety of options, such as initial deferred-applet mode where an initial image is displayed with a click on the image or link on the page initiating applet/canvas 3D modeling, and image+loading mode in which case the 3D model is loading behind the scenes while an initial image is displayed.

- **JavaScript Objects**

Creates a single JavaScript object, Jmol, which includes a set of functions and internal objects such as Jmol.Applet, Jmol.Image, and Jmol.controls.

- **JavaScript Prototypes**

The object you create using `Jmol.getApplet()` is a JavaScript object that is a subclass of `Jmol.Applet`. When you use `Jmol.getApplet()`, you get a reference to a JavaScript object, not the Java applet/canvas itself. The applet or canvas is wrapped in a set of div elements, allowing a richer diversity of options.

- **AJAX**

JSmol includes methods to easily access cross-platform resources using AJAX provided by jQuery.

- **Scripting**

JSmol provides the same full complement of scripting that Jmol offers. JSmol accepts script commands immediately, before or during applet/canvas creation on the page, caching them until Jmol is ready to accept them [60].

3.14.2 JSmol Initialization

JSmol makes good use of the HTML5 features. Consequently, it is well suited for only modern web browsers. JSmol works with Internet Explorer starting from version 9. Furthermore, it is essential to use a **doctype** in the header of the HTML page. Besides, for the sake of full compatibility, charset should be declared as UTF-8 encoding for localization i.e language translations of the JSmol pop-up menu. Also, the HTML document must be saved using UTF-8 encoding. For those reason, the HTML documents should start as [60]:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
```

The web page should have the following in the head section (pointing to appropriate paths if not the same folder as the web page as shown here):

```
<script type="text/javascript" src="JSmol.min.js"></script>
```

3.14.3 Setting Parameters

According to JSmol documentation, we can make the essential and minimal call to create a Jmol object using:

```
Jmol.getApplet("myJmol")
```

This will create a `myJmol` global variable in JavaScript that holds the Jmol object and is likewise the unique ID for that object in all functions and methods described below. It is pertinent to note that this syntax will work only when the HTML file is located in the root JSmol folder [60]. Nevertheless, the method for indicating variables is different. The call to create a Jmol object with specified characteristics is to define an `Info` variable, which is an associative array (a set of key+value pairs). This shows all the desired characteristics of the Jmol object. The Jmol-JSO library will provide a default `Info` variable, so we only need to specify those keys with their corresponding values that we want to customize. As soon as we have defined `Info`, we can then create and insert the Jmol object in the page using this [60]:

```
Jmol.getApplet("myJmol", Info)
```

Bear in mind that `myJmol` and `Info` are user-defined variables and may be given any name. `myJmol` becomes the identifier of the particular Jmol object that is being created. We may decide to have two Jmols on our page and call them `jmolA` and `jmolB` for instance. We may also use the same set of parameter `Info`, or use two different sets named `InfoA` and `InfoB` for example. In contrast, Jmol must be written

from the start as such, because it is the internal name and identification of the unique Jmol object constructor. For example [60]:

```
var Info = {  
    color: "#FFFFFF",  
    height: 300,  
    width: 300,  
    script: "load $caffeine",  
    use: "HTML5",  
    j2sPath: "j2s",  
    jarPath: "java",  
    jarFile: "JmolAppletSigned0.jar",  
    isSigned: true,  
    serverURL: "php/jsmol.php",  
    disableInitialConsole: true  
};  
  
Jmol.getApplet("myJmol", Info);
```

[60]

3.14.4 CSS 3.10.4

Cascading Style Sheets (CSS) is a language used for describing the presentation of a document written in a markup language [58]. we have used CSS to specify the application's web page colors, layout and fonts for in the application. On a general note, CSS also allows the modification of the display of such documents to different types of devices, such as large screens, small screens, or printers. CSS is independent

of HTML and can be used with any XML-based markup language. The separation of HTML from CSS makes it easy to maintain websites, share style sheets throughout pages, and modify pages to specific environments. This is called the separation of structure from presentation [58].

3.15 Integrated Development Environment

An Integrated Development Environment (IDE) is described as a software program that contains a series of tools such as code editor, compiler, debugger and many other useful tools needed to effectively write, and test, refactor code and generate diagrams for an application under development. Examples of IDEs are Emacs, Eclipse, Idea, JBuilder, Visual Studio, Netbeans, etc.

3.15.1 PyCharm

We have used PyCharm mainly because PyCharm is an IDE used in computer programming, specifically for the Python language [47]. It is developed by the Czech company JetBrains. It provides code analysis, a graphical debugger, an integrated unit tester, integration with VCS, and supports web development with Django. PyCharm is platform independent as it works Windows, Mac OS X and Linux versions. Other advantages of PyCharm IDE are [47]:

- Coding Assistance and Analysis: Comes with code completion, syntax and error highlighting, linter integration, and quick fixes.
- Project and Code Navigation: Renders specialized project views, file structure views and quick jumping between files, classes, methods and usages.

- Refactoring: including rename, extract method, introduce variable, introduce constant, pull up, push down and others.
- Support for web frameworks: Django, Web2py and Flask.
- Integrated Python Debugger: Integrated unit testing, with line-by-line coverage, offers Google App Engine Python development.
- Version Control Integration: unified user interface for Mercurial, Git, Subversion, Perforce and CVS with changelists and merge.

3.16 Version Control System

Version control system (VCS) on the other hand records changes to a file or set of files over time. In today's world of programming, choosing a VCS has numerous advantages ranging from reverting and tracking changes made to a file, and comparing changes over time. It has proven valuable in collaborations across borders and tracking errors. We have used git in this project.

Chapter 4

Application Specifications

4.1 Introduction

This chapter focuses on the specification of the CML application under development. It includes the requirements, specifications, use cases, and milestones for the project. Also, a brief introduction to CML is presented.

4.2 Chemical Markup Language

Chemical Markup Language (CML) is a method to managing chemical information with the use of tools such as XML and Java [43]. It was the first domain specific implementation strictly based on Xtensible Markup Language (XML). CML was first based on a Document Type Definition (DTD) [41], and later on an XML Schema [42]. XML Schema is one of the most powerful and extensively used system for information management. It has been developed over more than a decade by Murray-Rust, Rzepa and others [26, 33, 41, 42, 43] and has been tested in many areas, and on a variety of machines.

CML is capable of supporting a wide range of chemical concepts including:

- molecules
- reactions[26]
- spectra and analytical data[33]
- computational chemistry
- chemical crystallography and materials

4.3 CML Visualization Web Application

The CML visualization application is a web-based application predominantly built for chemistry researchers, professors, students and people of associated interest with the main functionality of providing 3D interactive visualization of CML molecular data in different models and their corresponding properties.

4.4 Primary Requirements

The application of TDD to the development of a visualization application is the focal point of this research. As required, the application is web-based. The users of this application will predominantly be chemistry students, professors, researchers, and people of associated interests from all over the world who in the cause of their research need to visualize molecular information from CML files containing raw information. With regards to the primary requirements, the CML visualization application is expected to have the following functionalities:

Upload Functionality

- Ability to upload a large CML tree containing initialization and finalization module of molecular data.
- Upload already extracted CML files.

Data Extraction

While handling the upload of a large CML tree, the application is expected to:

- Extract hundreds of specific CML files from the uploaded large CML tree.
- Extract specified parameters from the tree such as dipole vectors, task, Basis Set Information etc.

Storage Functionality

- Store only extracted small CML files from the large CML tree in the database.
- Discard the tree after extraction and storage.
- Create specified fields where extracted parameters are stored.

Database

The application is expected to have a database with specified fields for storage of the following information from the extracted small CML trees. The specified fields are:

- Filename(string).
- Small CML molecule (<molecule> element)(string or text or file).
- Task (e.g. geometry optimization, frequency) (string).
- Basis Set Label (string).

- Method (string).
- SMILES (string).
- InChI (string).
- Program name.
- Program version.

Visualization Functionality

- Visualize molecular geometry of the stored files in different visualization models: wireframe, ball and sticks and spacefill.
- Create buttons to trigger the correct Jmol script that shows the different visualizations.
- Visualize surfaces: Van-der-Waals Surface (VDW), Solvent Accessible Surface (SAS), transparent surface.
- Visualize dipole moment vector.
- Label partial charges: obtain the array with partial charges (one floating point value per atom) from uploaded files and load the data into JSmol.
- Map Electrostatic Potential (ESP): Once JSmol knows the partial charges, it can visualize the ESP either directly or mapped on VDW or SAS surfaces.
- Animate vibrations: JSmol can already display the vibrations read from many file types but not yet the displacement vectors from CML files. Load the vectors into the models.

Search

- Create a search functionality: Since millions of files will be uploaded to the application, there should be a search functionality.

4.5 Use Cases

From the requirements, a number of use cases of the CML visualization web application were determined. A mock up of the different specifications were made and the specifications were divided into different views based on the required functionality. These gave birth to the following views:

List view: Here a user can see uploaded, extracted and stored CML files from the already discarded large CML tree. At the same time, a user can see uploaded (externally extracted) CML files. The list view will have the following properties:

- Shows list of all entries (CML files) in the database.
- Selecting an entry should open it in single molecule item view.
- Selecting two or more entries should open them in multiple molecule item view.
- Has link to upload page.

Upload view: Here a user should be able to upload a large CML tree and have all molecules programmatically extracted from the large tree. This view should also afford a user the opportunity to upload an externally extracted CML molecule as well for visualization. This view will have:

- Form with file input box.

- Upload (submit) button.

Item view: Here a user can visualize the selected molecule from the list view. This can be divided into either single molecule item view, (a view of one visualized CML molecule), and an optional multiple molecule item view (visualizing two or three CML molecules side by side). This view will have:

- JSmol script shows molecule.
- Panel with control buttons.
- Radio buttons: balls and sticks, wireframe, spacefill.
- Show/hide dipole moment vector (toggle button).
- Partial charges as atom labels.
- Different surfaces (Van-der-Waals, Solvent Accessible Surface).
- Mapping Electrostatic Potential (ESP) on surface.
- Animate Vibrations.

With these initial specifications, milestones were determined in order to track project progress and to ensure that none of the requirements are missed.

4.6 Milestones

A milestone is a significant event in the course of a project that is used to give visibility of progress in terms of achievement of predefined goals [7]. Failure to meet a milestone indicates that a project is not proceeding according to plan and usually triggers corrective action by management. The specifications and the requirements

were further divided into milestone goals basically for more clarity and to ensure all requirements are reached during the course of the development while ensuring steady progress is made as well.

1. Django webapp with item view showing a single molecule with JSmol (e.g a file loaded from `/static/...`).
2. Add basic controls for visualization (balls and sticks, wireframe, spacefill).
3. Ability to upload a small CML file, store it (as string) in the database, add entry to list view, item view loads molecule structure from database (instead of `/static`).
4. Upload large CML file (instead of small) and while handling the uploaded data, extract the CML molecule (effectively the content of the small CML file) from the large XML tree. Only store the small CML molecule(s).
5. Also extract CML: task, store alongside the corresponding CML molecule in the database, and display in list and item view.
6. Extract dipole moment vector (three floats), store in database and use it to draw arrow in JSmol.
7. Extract net atomic charge (one float per atom), store in database and load this data as partial charges into JSmol molecule, add ESP surface.
8. Extract more textual data from CML: Method, Basis set label, SMILES, InChI.
9. Add animation for vibrations (only available when task is frequency_calculation).
10. Add visualization for molecular orbitals.
11. Make it look nice (CSS Stylesheet).

4.7 Design Visualization

In agile software development, informal design visualizations are often used by developers to ascertain that all the requirements are being captured. The most common of these informal drawings and sketches are low fidelity sketches. Furthermore, these drawings are also used to understand design decision [8, 55]. Understanding is important when developers try to make sense of a code snippet. It can be beneficial when training a new developer or when a developer needs to work with a colleague's code [8]. Rough sketches and drawings have been found to be invaluable in that situation to frame and try to understand a problem or some obscure architecture. Since sketches are also used to develop the UI and to brainstorm about the right architecture for the project, developers often collaborate to develop those sketches, and they save these visualizations because it serves as documentation [8, 55]. Informal drawings capturing the requirements and the design of the AUD in accordance with the initial specifications were determined, as illustrated by Figure 4.1 and 4.2.

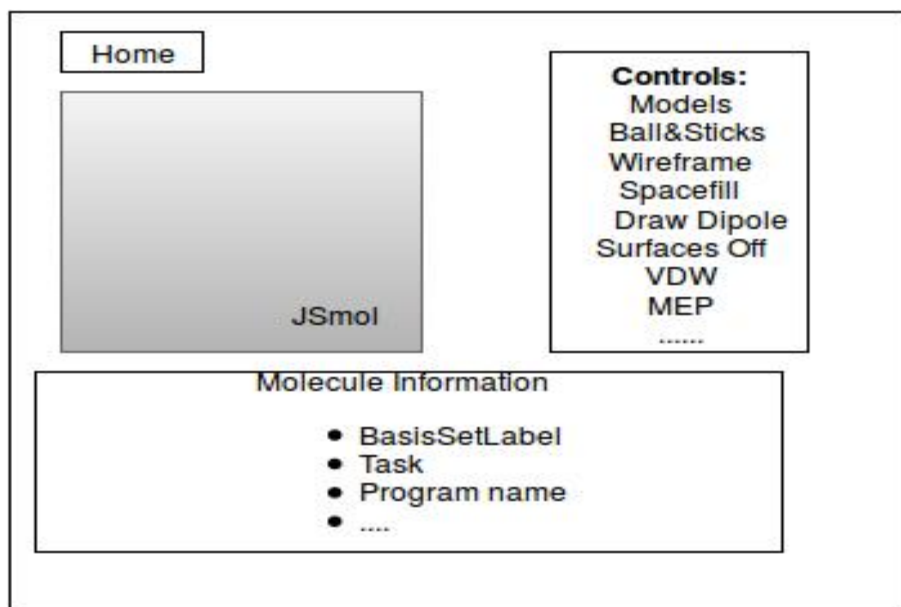


Figure 4.1: An informal drawing of the application with Single Molecule Item View.

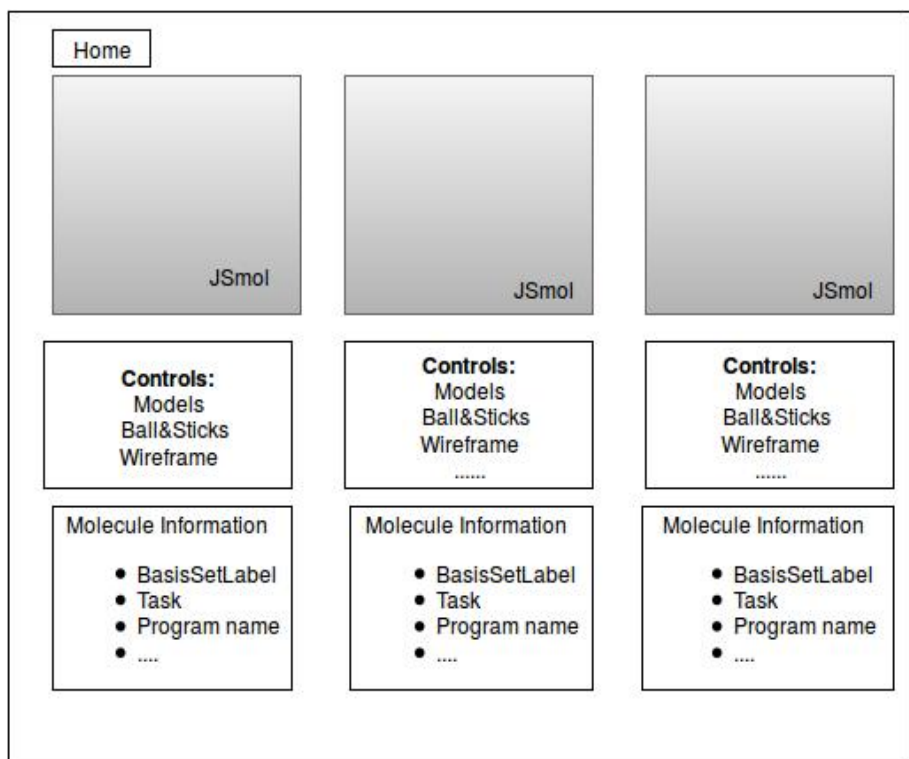


Figure 4.2: An informal drawing of the application with Multiple Molecule Item View.

4.8 Secondary Requirements

These additions are basically modifications to the design and additional implementation of the RESTful API for both the single molecule item view and the multiple molecule item view. The following are the secondary requirements:

List view:

- Forward `http://server:port/` to `http://server:port/list`.
- Make table look nicer. For example, table borders with thin grey horizontal dividers.
- Instead of storing the file-path (directory + filename), just store the filename.

- Make it more appealing with CSS3.

Item view:

Single molecule item view (see Figure 4.1)

- Implement RESTful interface for single molecule item view.
- Have single molecule item view under `//server:port/view`. For example, `http://server:port/1` loads molecule with ID 6.
- Add control for transparency of surfaces.

Multiple molecule item view (see Figure 4.2)

- Implement RESTful API for multiple molecule item view. For example `http://server:port/1` compares (database Ids) 2, 6 and 10.
- Have multiple molecule item view under `http://server:port/multiview`.
- Add sync control.
- Add control for transparency of surfaces.

Chapter 5

Implementation With TDD

5.1 Introduction

Having described the requirements in the previous chapter, this chapter discusses the implementation of our CML visualization application with the TDD approach. We give details into the technical information about the system, the structure of the system, and the interaction among the various tools used in building the application.

5.2 Application Overview and Design

The application was developed using Django WAF and it consists of several Python files that provide the required functionality. At the center of the design lies TDD which offers a lot of advantages in the literature. TDD starts with gathering requirements which are transformed into user stories. Those user stories are then translated into automated functional tests. The automated functional tests will drive the unit tests, which eventually becomes the production code.

The project began by setting up the environment and which ultimately starts with an automated functional test. The system was implemented incrementally according to TDD practice with one functionality implemented each step at a time. At every step we wrote a failing test, which leads us to code each unit of functionality as we refactor along the line. At first we had a working skeletal prototype, and later on we improved and continued to build on that until we had the final working version. In line with Django's architecture, we arrived at Figure 5.1 as the overall architecture of the CML visualization application with detailed explanation in Section 5.4.

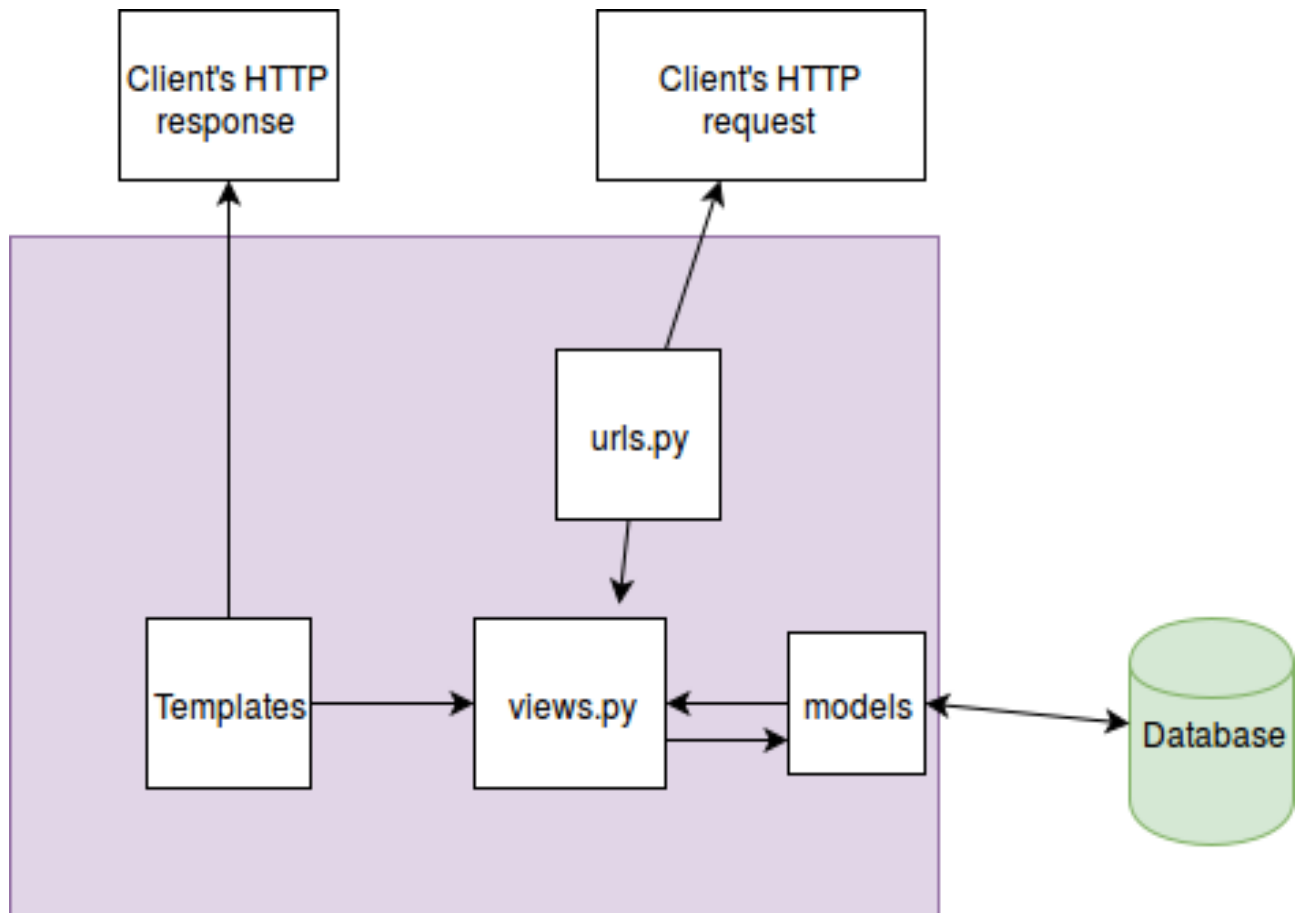


Figure 5.1: Overall application architecture

5.3 Project Structure

The application contains different directories with each of them serving collective and individual roles towards the overall functioning of our application as shown in Figure 5.2.

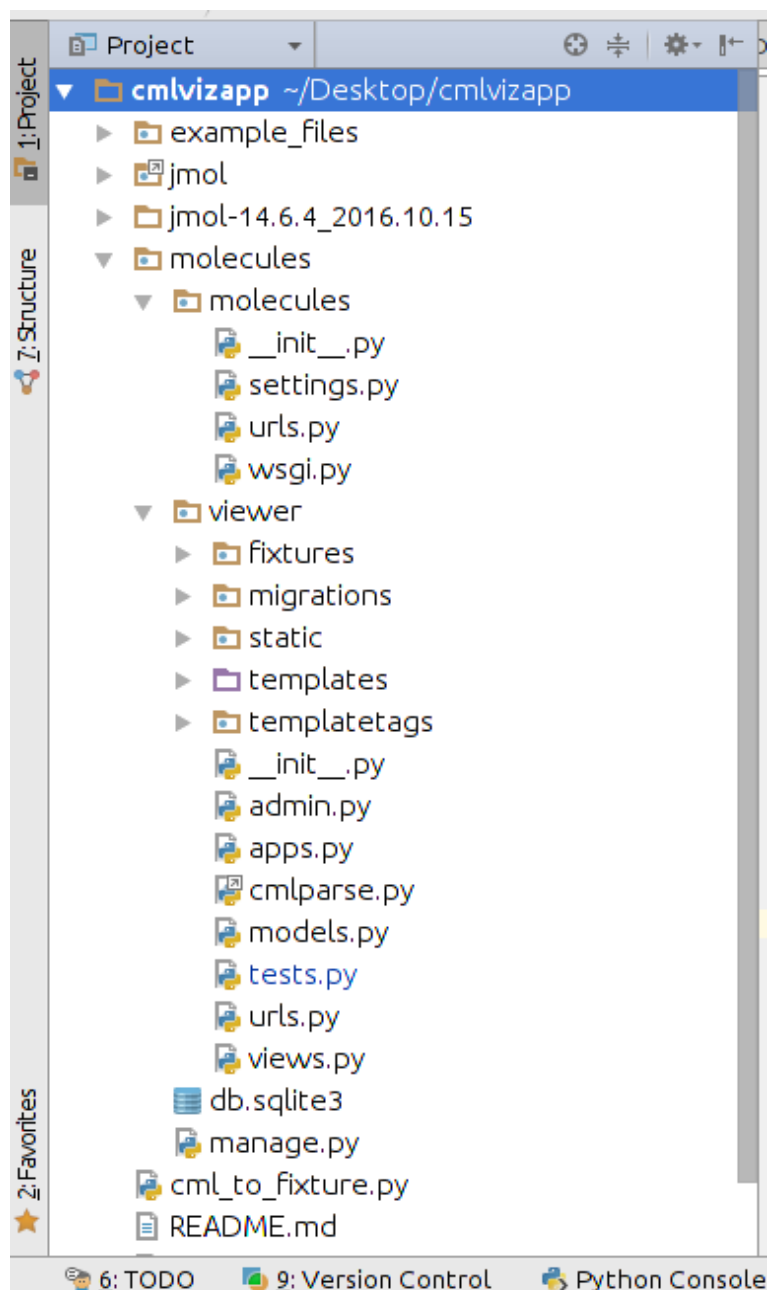


Figure 5.2: The Project directory structure containing the visualization application.

5.4 Explanation of Application Structure

The project directory contains different directories with various files that make up the whole application. A short overview is explained below:

- **cmlvizapp**: This is called the outer root directory. It is the container for the project and we could give it any name. This folder houses the entire application including the JSmol applet, and example files folder which contains the CML files.
- **molecules**: This directory contains the project settings like the `init.py` file, `settings.py`, `urls.py` and `wsgi.py`. An explanation of each of the aforementioned files was given in Section 2.6.
- **viewer**: This is the application directory. It was created using the convention `Python3 manage.py startapp viewer` as demonstrated in the sample application example. It contains the `fixtures`, `static`, `templates`, and `templatetags`. Fixtures are used to provide initial data to an application in Django. A fixture is a collection of data that Django knows how to import into a database.[11] Static contains the CSS file for the application and all other static assets used by the application. For example, JSmol and the CSS style sheet used to beautify the application.
- **templates** : This has two directories, namely public and the viewer directory. The public directory contains the `base.html` file, while the viewer directory contains `listing.html` (list view html code), `upload.html` (upload view html code), and `view.html` (item view html code). We implemented template inheritance here in order to avoid code duplication in our templates. These templates are rendered through the view. The `base.html` in the public directory inherits

the three templates from the viewer directory as seen in Figure 5.3.

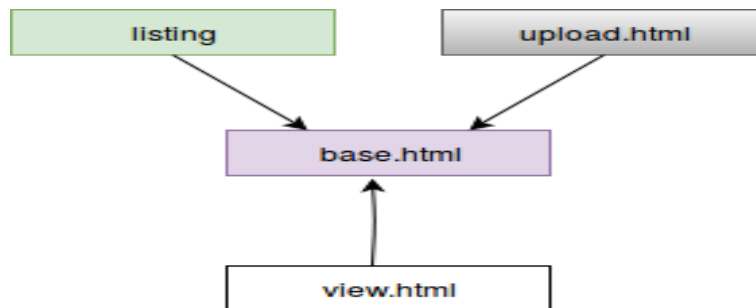


Figure 5.3: Template inheritance.

- **views.py** : Contains all the callable view functions of the applications. The view serve the correct templates when requested by the client through the browser. For example, running the command `python3 manage.py runserver` triggers our development server at the `http://127.0.0.1:8000/list`. Starting up the server automatically sends an HTTP request for the list view through our browser. The URL takes the request and match it to the appropriate view using URL resolver, and Django loads the appropriate view. Having served the list view, we can then select one or two entries (molecules) and click on the view button at the bottom of the list view page which calls `view.html` and responds with either the part of the code that displays a single molecule item view or a multi molecule item view as appropriate.
- **templatetags**: Consists of the filter functions. `Cmlparse.py` is a Python function that was written basically for the data extraction functionality, and to convert CML files to fixtures for providing initial data to Django. `Models.py` contains data registered and stored in the database.
- **tests.py** : This file has about 600 lines of code for our automated test framework with different classes. In TDD, the test drives the production code of the AUD.

- **urls.py** : Contains the overall URL configurations as mapped to the views of the application.

5.5 Setting Up Application Functional Tests

The first step in TDD is writing functional tests that keep failing, getting those tests to pass, and running them iteratively as we build our application with the user story (requirements). So we get started from the `test.py`. We start by importing some modules. We discuss here only the most important modules used in writing the automated test cases for the application.

5.5.1 Django TestCase

This is the most popular class used for writing tests in Django. It inherits from `TransactionTestCase` (and by extension `SimpleTestCase`). `SimpleTestCase` is suitable for an application that does not use a database [15].

The Django `TestCase` class [15]:

- Wraps the tests within two nested `atomic()` blocks: one for the entire class and one for each test. As a result, `TransactionTestCase` is well suited for testing specific database transaction behaviour.
- Verifies postponed database constraints at the termination of each test[15].

Therefore as part of the important package for testing we called Django `TestCase` in line 13 of Figure 5.4.

```
from django.test import TestCase
```


5.5.2 Selenium WebDriver

Selenium WebDriver is an automation tool for activities in a browser [50]. A detailed explanation of the Selenium package has been given in Section 3.8. Line 16 to 18 in Figure 5.4, import all the Selenium modules necessary to run our test. At this juncture, it is pertinent to explain that according to TDD practice, test methods are named to clearly specify what they do. Also, the docstring (Python documentation strings) should give clear details as to what the test does.

```
1  """
2  This program does functionality and drives Unit test
3  for CML Visualization Application Using Selenium,
4  HTML5, Javascript, JSmol Script Python3, Django and RESTful API.
5
6  References
7  [1] https://lincolnloop.com/blog/introduction-django-selenium-testing/
8
9  [2] https://docs.djangoproject.com/en/1.10/topics/testing/tools/
10 """
11 import os, time, logging, re
12 from urllib import parse as urlparse
13 from django.test import TestCase
14 from django.contrib.staticfiles.testing import StaticLiveServerTestCase
15 from django.contrib.auth.models import User
16 from selenium.webdriver.support.ui import WebDriverWait
17 from selenium.webdriver.support.expected_conditions import staleness_of
18 from selenium.webdriver.firefox.webdriver import WebDriver
19 from django.template.tags.static import static
20 from selenium.webdriver.common.keys import Keys
21 from .models import Molecule
22 from .cmlparse import convert
23 logging.basicConfig(level=logging.DEBUG if __debug__ else logging.INFO)
24
25 ADMIN = 'admin'
26 PASSWORD = 'insecure'
27 EMAIL = 'root@localhost.localnet'
28 DJANGO_TEST_DELAY = int(os.getenv('DJANGO_TEST_DELAY', 0) or 0)
29 # TIMEOUT is number of seconds to implicitly\_wait\(\) for elements to appear
30 TIMEOUT = 30 if __debug__ else 10
31 MOLECULES = os.path.join(os.path.dirname(__file__), 'static', 'molecules')
32 TESTFILE = os.path.join(os.path.dirname(__file__), '..', '..',
33                          'example_files', 'CML_CompChem', 'GS_H2O_B3LYP_631G.cml')
34
```

Figure 5.4: Importing modules and setting variables.

Having imported the necessary modules and `WebDriver`, we then set up the paths and assigned them to a variable as illustrated in figure 5.4. The next step is testing the URL. We start with the URL because it is an important component of the application that the users come in contact with. Python is an object oriented programming language which means that there is a construct called a class that enables structuring software in a particular way. With classes, we add consistency to our framework so that they can be used in a cleaner way [51].

5.5.3 Opening Page with WebDriver

We get started with Selenium by sub-classing `WebDriver` to create our own test case, `TestWebDriver`, and our functions (see Figure 5.5 continuation). The function "open" gets the URL so we could test it. Then we wrote the the last function `wait_for_page_load` that waits for the page to load. All the functions were named in such a way that they are self explanatory and can serve as documentation according to TDD practice.

```

36 class TestWebDriver(WebDriver):
37     """
38     Trivial subclass of WebDriver that adds some features
39
40     http://www.obeythetestinggoat.com/
41     how-to-get-selenium-to-wait-for-page-load-after-a-click.html
42     """
43     current_page = None
44     _current_url = None
45
46
47     def __init__(self, *args, **kwargs):
48         """
49         Override __init__ to add some options
50         """
51         super(TestWebDriver, self).__init__(*args, **kwargs)
52         self.implicitly_wait(TIMEOUT)
53
54     @property
55     def current_url(self):
56         """
57         Override property of WebDriver
58         """
59         driver_url = super(TestWebDriver, self).current_url
60         logging.debug('WebDriver current_url: %s', driver_url)
61         if driver_url == 'about:blank':
62             driver_url = ''
63         return driver_url or self._current_url

```

Figure 5.5: A code fragment that demonstrate the opening of page with the class TestWebDriver.

Later, we used Xpath to locate element in Figure 5.5 (continuation). Elements in a web page context are simply discreet components of an HTML document or web page. Element types could range from: heading, table, images. A detailed explanation as to how XPath works is given in Section 5.6.

```

64
65 @current_url.setter
66 def current_url(self, url):
67     """
68     Make it writable
69     """
70     self._current_url = url
71     logging.debug('setter: current_url now %s', self._current_url)
72
73
74 def open(self, url):
75     """
76     Simplify opening by URL
77     """
78     logging.debug('Opening partial URL %s', url)
79     full_url = urlparse.urljoin(self.current_url, url)
80     logging.debug('Full URL is %s', full_url)
81     self.get(full_url)
82     self.current_page = self.find_element_by_tag_name('html')
83     time.sleep(DJANGO_TEST_DELAY)
84
85
86
87 def wait_for_page_load(self, timeout=30):
88     WebDriverWait(self, timeout).until(staleness_of(self.current_page))
89     self.current_page = self.find_element_by_tag_name('html')
90     time.sleep(DJANGO_TEST_DELAY)

```

Figure 5.5 (cont.) : Continuation of Figure 5.5.

5.6 Locating Elements

There are various strategies to locate elements in a page. Selenium provides the following methods for this purpose [17]:

- `find_element_by_id`
- `find_element_by_name`
- `find_element_by_link_text`
- `find_element_by_tag_name`

- `find_element_by_class_name`
- `find_element_by_css_selector`
- `find_elements_by_partial_link_text`
- `find_elements_by_xpath`

5.6.1 Locating by XPATH

XPath is a powerful language used for locating nodes in an XML document, [17] and it was used extensively in the AUD. With Selenium, we can take advantage of XPath to locate elements in web applications as HTML can be an implementation of XML (XHTML). XPath capabilities extends far beyond just finding elements, as it can also be used to locate third checkbox on a page. One of the core strength of XPath is when a user does not have a suitable `id` or `name` attribute for the element to be located, XPath can still be used to locate the element either absolutely or relatively to an element that does have an `id` or `name` attribute. XPath locators can also be used to specify elements via attributes other than `id` and `name` [17].

5.6.2 StaticLiveServerTestCase

In accordance to Django’s documentation [14], when running tests that use real HTTP requests as a substitute for the built-in testing client `LiveServerTestCase`, the static assets need to be served along with the rest of the content. This is done, so that the test environment looks exactly like the actual one as precisely as possible. However, `LiveServerTestCase` has very basal static file-serving capability. It’s not aware of the `finders` attribute of the `staticfiles` application and presumes that the static content has already been collected under `STATICROOT`.

As a result, `staticfiles` ships its own `StaticLiveServerTestCase`, a subclass of the built-in one that has the capacity to serve all the assets during execution of these tests in a very similar way to what we get at development time with `DEBUG = True`, i.e. without having to collect them using `collectstatic` first [14]. This is the reason why we needed to import `StaticLiveServerTestCase` on line 14 in Figure 5.4. With `StaticLiveServerTestCase`, we have imported a slightly advanced development server, therefore we need to create a function with our selenium automation tool that will open and close the browser.

5.6.3 Set Up and Tear Down

In Figure 5.6, we are subclassing `StaticLiveServerTestCase` to create our own test case `SeleniumTestCase`, setting up the web driver in `setUp()` which loads the browser, and closing the browser after the tests are run with `tearDown()`. The `setUp()` function is a method from the `TestCase` superclass that runs before each test method.

```
92
93
94 class SeleniumTestCase(StaticLiveServerTestCase):
95     """
96     A base test case for Selenium, providing helper methods for generating
97     clients and logging in profiles.
98     """
99     def setUp(self):
100         """
101         setUp is where you setup call fixture creation scripts
102         and instantiate the WebDriver, which in turns loads up the browser.
103         """
104         self.webdriver = TestWebDriver()
105         self.webdriver.current_url = self.live_server_url
106
107     def tearDown(self):
108         """
109         Don't forget to call quit on your webdriver, so that
110         the browser is closed after the tests are run
111         """
112         self.webdriver.quit()
```

Figure 5.6: Setup and teardown browser

In Figure 5.7 below, we created a Django admin sign-in test class that inherits from the base class. This class has two functions, namely: `setUp()`, and `skip_test_login`. The `setUp` was customized for authorization, while the `skip_test_login` tests the homepage by locating specified element in the page. Then we set Django admin log-in test before we move into implementing the requirements one after the other. Having laid the foundation, we can start our functional test with Selenium as we implement the test cases first and then code the features in the `views.py` and other dependent files as appropriate.

```
116
117 # Make sure your class inherits from your base class
118 class Auth(SeleniumTestCase):
119     """
120     Login to admin page
121     """
122     def setUp(self):
123         """
124         Customized setUp for authorization
125         """
126         super(Auth, self).setUp()
127         User.objects.create_superuser(username=ADMIN,
128                                     password=PASSWORD,
129                                     email=EMAIL)
130
131     # Just like Django tests, any method that is a Selenium test should
132     # start with the "test_" prefix.
133     def skip_test_login(self):
134         """
135         Django Admin login test
136         """
137         # Open the admin index page
138         self.webdriver.open('/admin')
139         self.webdriver.find_element_by_name('username').send_keys(ADMIN)
140         self.webdriver.find_element_by_name('password').send_keys(PASSWORD)
141         self.webdriver.find_element_by_xpath('//input[@value="Log in"]').click()
142         self.webdriver.wait_for_page_load()
143         # make sure we logged in by looking for something not on login page
144         text = self.webdriver.find_element_by_tag_name('body').text
145         logging.debug('text: %s', text)
146         self.assertIn('WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT',
147                     text)
```

Figure 5.7: The setting up of Django admin log in test

After everything is set, we then start implementing the functionality one step at a time in the TDD manner. We start with implementation of the list view as given in Section 5.7.

5.7 Implementing List View

The convention for implementation of each of the specified views follows the same pattern according to the TDD principles as explained in Chapter 2. First we write a failing test in the `tests.py` file, then write the minimum code to get the test to pass, update the `views.py` as well as the `urls.py` ensuring it maps to the right view function, and extend the test cases as we code each of the specifications. We update the corresponding HTML files to be served by `views.py` from the template and we refactor as we go.

To get the list view implemented, we started by writing a class and delegating appropriate instructions to the function on Line 157 of Figure 5.8. Python functions are a convenient way to divide code into useful blocks, allowing us to order our code, make it more readable, reuse it and save some time [57]. The test case in Figure 5.8 tested and verified the items related to the list view as specified. It is pertinent to note that during the writing of each of the test cases, some debugging was done.


```

152
153 class Views(SeleniumTestCase):
154     """
155     Test views as specified in first specification doc
156     """
157     def test_listing_view(self):
158         """
159         Specifications:
160         (1) Shows list of all entries (molecules) in the database
161         (2) Clicking on an entry shows it in item view
162         (3) Has link to upload page
163
164         This test case will only test items 1 and 3. `test_display_view`
165         will test item 2.
166
167         Added in second specification doc list: make sure `/` redirects to `/list`
168         """
169
170         logging.debug('starting test_listing_view')
171         self.webdriver.open('/')
172         self.assertEqual(urlparse.urlsplit(self.webdriver.current_url).path,
173                          '/list')
174         header = self.webdriver.find_element_by_xpath('//tr[1]')
175         logging.debug('text: "%s"', header.text)
176         self.assertEqual(header.text, "Select ID Name Source File Index Task")
177         link = self.webdriver.find_element_by_link_text(
178             "Upload a local CML file")
179         logging.debug('link: "%s", "%s"', link.tag_name, link)
180         self.assertEqual(link.tag_name, 'a')
181         return link # for test_upload_view
182

```

Figure 5.8: The test written for listing view.

The code for the listing view in Figure 5.9 and 5.9 (cont.) below was written and updated simultaneously as driven by the test in Figure 5.8. We defined three functions that cater to the display of molecules either in single molecule item view or multiple molecule item view in accordance with the specification.

```

1  import sys, os, django, operator
2  if os.path.splitext(os.path.basename(sys.argv[0]))[0] == 'pydoc':
3      django.setup()
4  from django.shortcuts import render, redirect
5  from django.db.models import Q
6  from functools import reduce
7  from django.http import HttpResponse, Http404
8  from .models import Molecule
9  from . import cmlparse
10 import logging
11 logging.basicConfig(level=logging.DEBUG if __debug__ else logging.INFO)
12
13 def listing(request):
14     """
15     List view of molecules
16     """
17     molecules = Molecule.objects.all()
18     fields = [field.name for field in Molecule._meta.get_fields()
19               if isinstance(field, django.db.models.fields.CharField)]
20     logging.debug('fields: %s', repr(fields))
21     if request.POST:
22         request.POST._mutable = True # allows us to modify it
23         logging.debug('Found input: %s', dict(request.POST))
24         if request.POST.get('display', '') == 'View Items':
25             request.POST['searchbox'] = '' # clear search box
26             selected = request.POST.getlist('selected')
27             if selected:
28                 if len(selected) == 1:
29                     return redirect('/view/%s' % selected[0])
30                 else:
31                     return redirect('/multiview/%s' % '+'.join(selected))
32             else:
33                 logging.info('"View Items" pushed but nothing found selected')
34         else:
35             searchfor = request.POST.get('searchbox', '')
36             if searchfor:
37                 logging.debug('searching for: %s', searchfor)
38                 search = [(m + ':').split(':', 2)][2]
39                 for m in searchfor.split():
40                     logging.debug('search: %s', repr(search))
41                     filters = dict([(k + '__contains', v) for k, v in search if v])
42                     words = [k for k, v in search if not v]
43                     molecules = molecules.filter(**filters)
44                     for word in words:
45                         filters = [{k + '__contains': word} for k in fields]
46                         logging.debug('filters: %s', filters)
47                         q = (Q(**f) for f in filters)
48                         logging.debug('q: %s', q)
49                         molecules = molecules.filter(reduce(operator.or_, q))
50                         logging.debug('query: %s', molecules.query)
51     return render(
52         request,
53         'viewer/listing.html',
54         {'molecules': molecules, 'posted': request.POST, 'fields': fields})

```

Figure 5.9: The implementation of listing view in views.py.

```

56
57 def view(request, items=''):
58     """
59     Visual display of molecules using JSmol
60     """
61     selected = items.split('+')
62     logging.debug('selected: %s', repr(selected))
63     molecules = [Molecule.objects.get(id=s)
64                  if s.isdigit() else s for s in selected]
65     logging.debug('molecules: %s', repr(molecules))
66     if len(selected) > 1:
67         logging.debug('using multiview')
68         width = 100 / len(selected) - 5
69         div_width = width
70         side = min(35, width)
71         template = 'multiview'
72     else:
73         logging.debug('using singleview')
74         div_width = 96
75         side = 40
76         template = 'singleview'
77     return render(
78         request,
79         'viewer/view.html',
80         {
81             'selected': molecules,
82             'div_width': div_width,
83             'side': side,
84             'template': template,
85         }
86     )
87
88 def load(request, item):
89     """
90     Return data for a single molecule
91
92     This is a "helper" view, used by `view` indirectly, as the JavaScript
93     loaded by `view` then uses JSmol to load the molecule data from the
94     database, e.g.: `load load/0.cml`. This function is then called through
95     the URL dispatcher.
96
97     This is tested, therefore, by Views.test_display_view, and doesn't need
98     its own unit test.
99     """
100     if item.isdigit():
101         molecule = Molecule.objects.get(id=int(item))
102         return HttpResponse(molecule.data, content_type='text/plain')
103     else:
104         return redirect('/static/jsmol/data/%s' % item)
105

```

Figure 5.9 (cont.) : The continuation of Figure 5.9.

At the center of the implementation is the application `urls.py` file in Figure 5.10. The file uses regular expressions to map in-coming HTTP requests to the appropriate view.

```

1  from django.conf.urls import url
2  from django.views.generic import RedirectView
3  from . import views
4  urlpatterns = [
5      url(r'^list$', views.listing, name='listing'),
6      url(r'^$', RedirectView.as_view(pattern_name='listing', permanent=True)),
7      url(r'^load/(?P<item>.*)$', views.load, name='load'),
8      url(r'^upload$', views.upload, name='upload'),
9      url(r'^view/(?P<items>.*)$', views.view, name='view'),
10     url(r'^multiview/(?P<items>.*)$', views.view, name='multiview'),
11 ]

```

Figure 5.10: The URL configuration of our application.

The view uses the templates to respond by supplying the appropriate HTML response as seen in Figure 5.11 below:

```

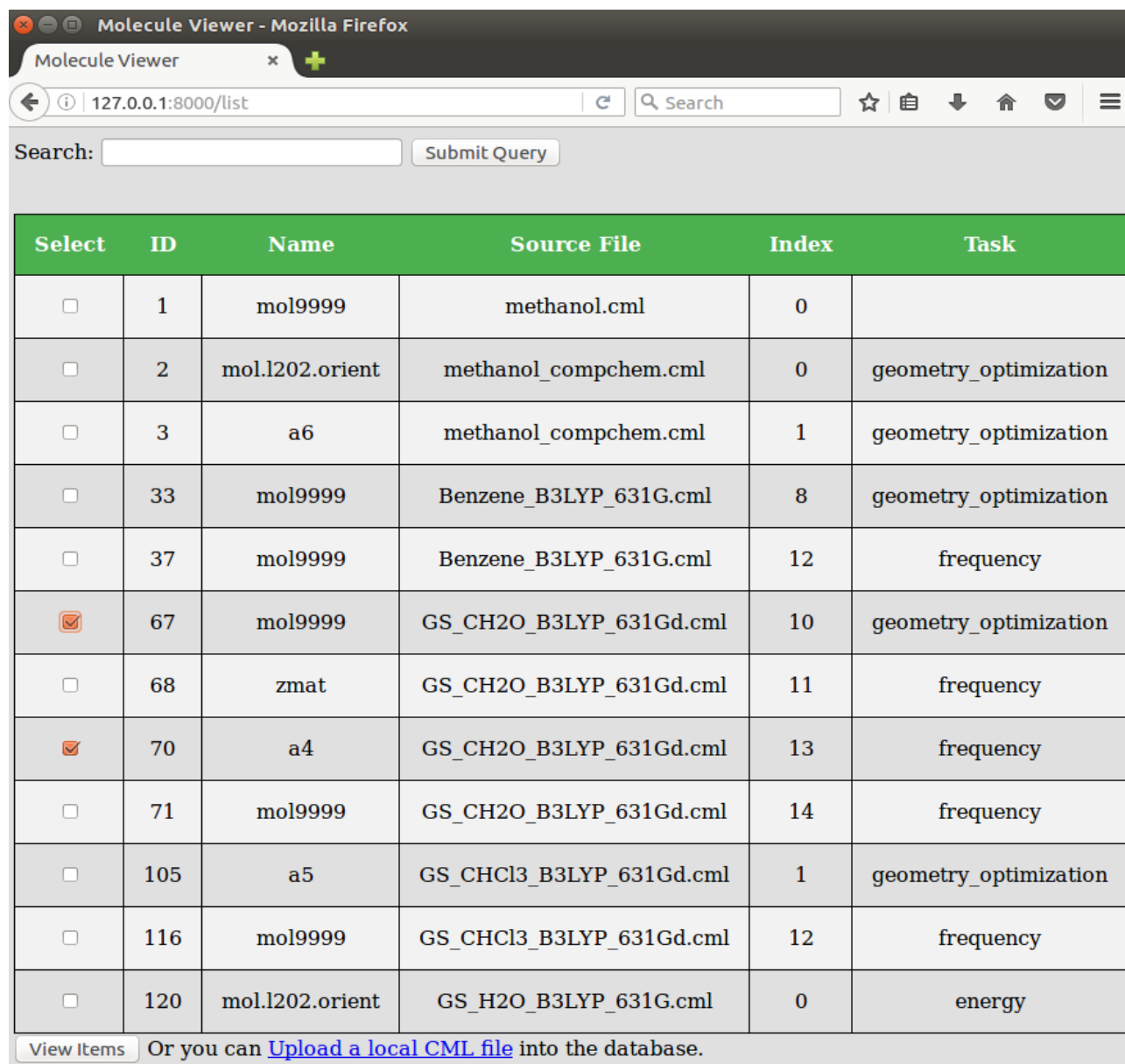
1  {% extends 'public/base.html' %}
2  {% block body_block %}
3  <form method="post">
4  {% csrf_token %}
5  <div id="searchwidget">
6      <label for="searchbox">Search:</label>
7      <input type="text" id="searchbox" name="searchbox"
8          value="{{posted.searchbox}}"/>
9      <input type="submit" id="search" name="search"/>
10     <br>
11     <br>
12     <br>
13 </div>
14 <table>
15 <tr>
16     <th>Select</th>
17     <th>ID</th>
18     <th>Name</th>
19     <th>Source File</th>
20     <th>Index</th>
21     <th>Task</th>
22 </tr>
23 {% for molecule in molecules %}
24 <tr>
25     <td><input type="checkbox" name="selected"
26         value="{{molecule.id}}"/>
27     <td>{{molecule.id}}</td>
28     <td>{{molecule.name}}</td>
29     <td>{{molecule.source}}</td>
30     <td>{{molecule.index}}</td>
31     <td>{{molecule.task}}</td>
32 </tr>
33 {% endfor %}
34 </table>
35
36 <a name="latest"><input type="submit" name="display" value="View Items"></a>
37 Or you can <a href="{{url 'upload' }}">Upload a local CML file</a>
38 into the database.
39

```

Figure 5.11: Listing view HTML code fragment.

5.7.1 List View

Running our application gives us the list view in accordance with the specifications. It also has an upload link and a navigation button to item view as shown in Figure 5.12.



The screenshot shows a web browser window titled "Molecule Viewer - Mozilla Firefox". The address bar displays "127.0.0.1:8000/list". Below the browser window, there is a search bar with the text "Search:" and a "Submit Query" button. The main content area contains a table with the following columns: Select, ID, Name, Source File, Index, and Task. The table lists 12 items, each with a checkbox in the "Select" column. Items 67 and 70 are selected. Below the table, there is a "View Items" button and a link to "Upload a local CML file into the database".

Select	ID	Name	Source File	Index	Task
<input type="checkbox"/>	1	mol9999	methanol.cml	0	
<input type="checkbox"/>	2	mol.l202.orient	methanol_compchem.cml	0	geometry_optimization
<input type="checkbox"/>	3	a6	methanol_compchem.cml	1	geometry_optimization
<input type="checkbox"/>	33	mol9999	Benzene_B3LYP_631G.cml	8	geometry_optimization
<input type="checkbox"/>	37	mol9999	Benzene_B3LYP_631G.cml	12	frequency
<input checked="" type="checkbox"/>	67	mol9999	GS_CH2O_B3LYP_631Gd.cml	10	geometry_optimization
<input type="checkbox"/>	68	zmat	GS_CH2O_B3LYP_631Gd.cml	11	frequency
<input checked="" type="checkbox"/>	70	a4	GS_CH2O_B3LYP_631Gd.cml	13	frequency
<input type="checkbox"/>	71	mol9999	GS_CH2O_B3LYP_631Gd.cml	14	frequency
<input type="checkbox"/>	105	a5	GS_CHCl3_B3LYP_631Gd.cml	1	geometry_optimization
<input type="checkbox"/>	116	mol9999	GS_CHCl3_B3LYP_631Gd.cml	12	frequency
<input type="checkbox"/>	120	mol.l202.orient	GS_H2O_B3LYP_631G.cml	0	energy

[View Items](#) Or you can [Upload a local CML file](#) into the database.

Figure 5.12: The display of listing view showing uploaded CML test files and their corresponding information.

5.8 Implementing Upload View

We implemented the upload view with Django's documentation on file uploads. Django handles a file upload by placing it in `request.FILES` [16]. We created a form that has a `post` method and a `multiple` attribute was specified in the `upload.html` file that resides in the template. The reason for having a `multiple` attribute is because we want the application to have the ability to upload multiple files in-line with the specification. The `upload` function in `views.py` handles the form and receives the file data in `request.FILES`. This was implemented by writing the test first before implementation. While Figures 5.13 and 5.14 show the test, Figures 5.15 and 5.16 show the implementation and the served HTML code respectively.

```
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216

def test_upload_view(self):
    """
    Specifications:
    Form with:
        (1) File input box
        (2) Upload (submit) button
    And from Milestones:
        (3) Ability to upload a "small" CML file, store it
            (as string?) in the DB, add entry to list view; item view
            loads molecule structure from DB (instead of /static)
        (4) Upload "large" CML/CompChem file (instead of "small")
            and - while handling the uploaded data -
            extract the CML molecule (effectively the content of
            the "small" CML file) from the larger XML tree.
            Only store the small CML molecule(s).
    """

    logging.debug('starting test_upload_view')
    self.upload_molecule_file('methanol.cml')
    name = self.webdriver.find_element_by_xpath('//tr[last()]/td[4]')
    logging.debug('name %s: %s', name, name.text)
    self.assertEqual(name.text, 'methanol.cml')
    'and now we upload a "big" cml file'
    self.upload_molecule_file('methanol_compchem.cml')
    index = self.webdriver.find_element_by_xpath('//tr[last()]/td[5]')
    logging.debug('index %s: %s', index, index.text)
    self.assertEqual(index.text, '22')
```

Figure 5.13: The test written for upload view.

```

367 def upload_molecule_file(self, filename):
368     """
369     This helper function. Upload a static file into the 'Upload' page.
370     """
371     link = self.test_listing_view()
372     link.click()
373     self.webdriver.wait_for_page_load()
374     input_element = self.webdriver.find_element_by_id('upload')
375     input_element.send_keys(os.path.join(MOLECULES, filename))
376     self.webdriver.find_element_by_name('process').click()
377     self.webdriver.wait_for_page_load()

```

Figure 5.14: An helper function for upload view.

```

L07 def upload(request):
L08     """
L09     Upload a small or large (multiple molecules) CML file
L10     """
L11     if request.POST:
L12         logging.debug('Found POST data: %s', dict(request.POST))
L13         logging.debug('FILES: %s', repr(request.FILES))
L14         try:
L15             files = request.FILES.getlist('upload[]')
L16             logging.debug('files: %s', repr(files))
L17             data = cmlparse.convert('', files)
L18             for item in data:
L19                 logging.debug('item: %s', item)
L20                 Molecule(**item['fields']).save()
L21             return redirect('/#latest')
L22         except Exception as problem:
L23             logging.warn('Problem uploading: %s', str(problem))
L24             raise
L25     return render(request, 'viewer/upload.html')
L26
L27

```

Figure 5.15: The implementation of upload view.


```

form
1  {% extends 'public/base.html' %}
2  {% block body_block %}
3    <form method="post" enctype="multipart/form-data">
4    {% csrf_token %}
5    <label for="upload">Upload CML file(s):</label>
6    <input type="file" name="upload[]" id="upload" multiple/>
7    <input type="submit" name="process" value="Upload"/>
8  </form>
9  {% endblock %}
10 {% comment %}
11 vim: tabstop=8 expandtab shiftwidth=2 softtabstop=2
12 {% endcomment %}

```

Figure 5.16: Upload view HTML code fragment.

5.8.1 Upload View

Running our application and clicking on browse button takes us to the upload view where a user can upload CML files (as shown in Figure 5.17).

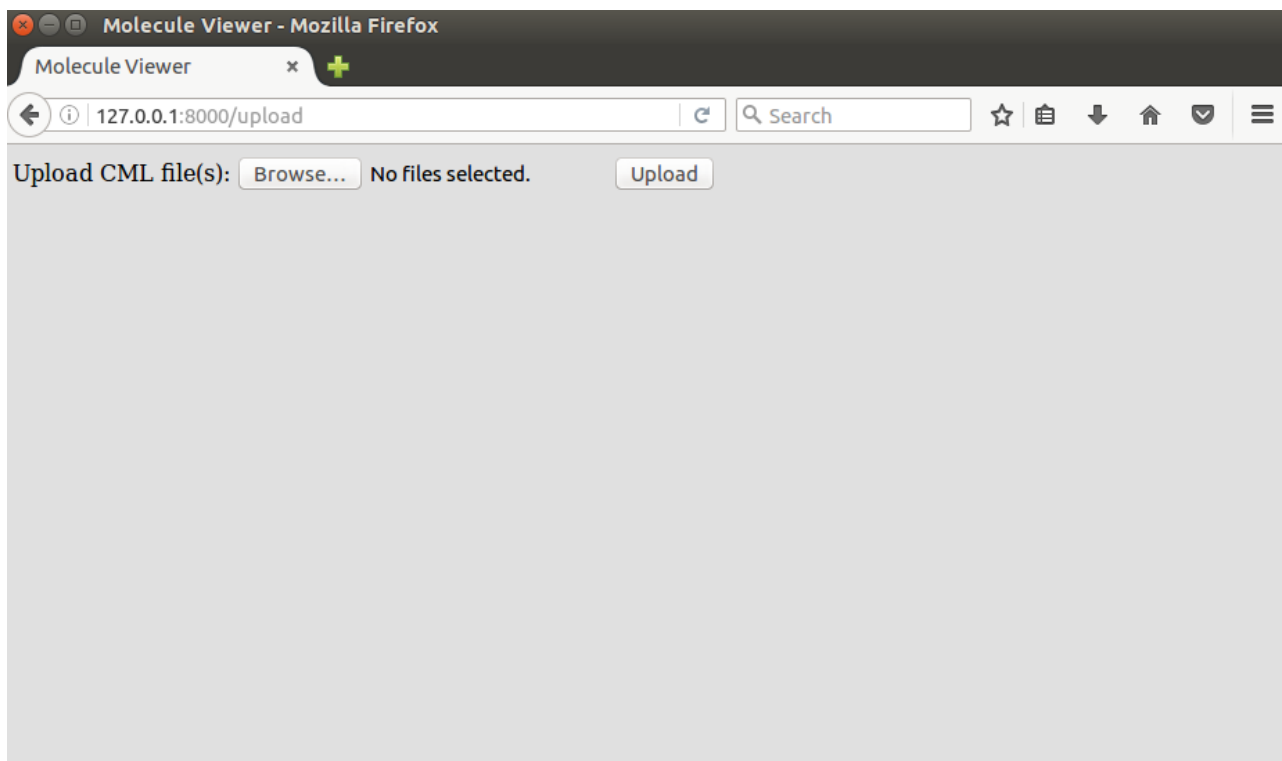


Figure 5.17: The display of upload view with the browse and upload button.

5.9 Implementing Item View

The item view is one of the key components of all the views as it holds the key to the visualization functionality of the application. As usual it started with writing a failing test which leads us to the iterative implementation as seen in Figure 5.18 and Figure 5.18 continuation.

```
222 def test_display_view(self):
223     """
224     Test the 'Item' (item display) view
225
226     Don't forget we still need to test, from test_listing_view(),
227
228     (2) Clicking on an entry shows it in item view
229
230     Specifications:
231     (1) JSmol shows molecule
232     (2) Panel with control buttons
233     (3) Radio buttons: balls & sticks, wireframe, spacefill
234     (4) Show/hide dipole moment vector (toggle button)
235     (5) Partial charges as atom labels
236     (6) Different surfaces (Van-der-Waals, Solvent Accessible Surface)
237     (7) Mapping electrostatic potential (ESP) on surface
238     (8) Animate vibrations
239     (9) Box with textual information from the database
240         (a) Filename
241         (b) Task
242
243
244
245 logging.debug('starting test_display_view')
246 self.upload_molecule_file('methanol.cml')
247 check = self.webdriver.find_element_by_xpath('//tr[last()]/td[1]/input')
248 check.click()
249 self.webdriver.find_element_by_name('display').click()
250 self.webdriver.wait_for_page_load()
251 'assume if canvas present, the molecule is being imaged'
252 canvas = self.webdriver.find_element_by_xpath('//canvas')
253 logging.debug('canvas: %s', canvas.get_attribute('outerHTML'))
254 radiobutton_boxes = self.webdriver.find_elements_by_xpath(
255     '//input[@type="radio"]/..')
256 logging.debug('radiobutton_boxes: %s', repr(radiobutton_boxes))
257 labels = ['ball and stick', 'wireframe', 'spacefill',
258           'van der waals', 'solvent accessible', 'surface off']
259
260
261 for box in radiobutton_boxes:
262     label = box.find_element_by_xpath('.//label')
263     labels.remove(label.text)
264 logging.debug('labels: %s', repr(labels))
265 self.assertEqual(labels, [])
266 'check for dipole vector checkbox'
267 dipole_checkbox = self.webdriver.find_element_by_xpath(
268     '//input[contains(@name, "dipole_toggle")]')
269 logging.debug('dipole_checkbox: %s', dipole_checkbox.get_attribute(
270     'outerHTML'))
271 dipole_checkbox.click()
272 logging.warn('*** VERIFY VISUALLY the dipole arrow was drawn ***')
273 time.sleep(3 if __debug__ else 1)
274 dipole_checkbox.click()
275 logging.warn('*** VERIFY VISUALLY the dipole arrow was deleted ***')
276 time.sleep(3 if __debug__ else 1)
277 'check for partial charges checkbox'
```

Figure 5.18: The test written for Item view.

```

278 charges_checkbox = self.webdriver.find_element_by_xpath(
279     '//input[contains(@name, "charges_toggle")]')
280 logging.debug('charges_checkbox: %s', charges_checkbox.get_attribute(
281     'outerHTML'))
282 charges_checkbox.click()
283 logging.warn('*** VERIFY VISUALLY partial charges are labeled ***')
284 time.sleep(3 if __debug__ else 1)
285 charges_checkbox.click()
286 logging.warn('*** VERIFY VISUALLY partial charges labels vanished ***')
287 time.sleep(3 if __debug__ else 1)
288 mep_checkbox = self.webdriver.find_element_by_xpath(
289     '//input[contains(@name, "mep_toggle")]')
290 logging.debug('mep_checkbox: %s', mep_checkbox.get_attribute(
291     'outerHTML'))
292 vdw_button = radiobutton_boxes[3].find_element_by_xpath('./input')
293 logging.debug('vdw_button: %s', vdw_button.get_attribute(
294     'outerHTML'))
295 vdw_button.click() # Van der Waals surface
296 mep_checkbox.click()
297 logging.warn('*** VERIFY VISUALLY electrostatic potential mapped ***')
298 time.sleep(7 if __debug__ else 2)
299 mep_checkbox.click()
300 logging.warn('*** VERIFY VISUALLY ESP colors gone from surface ***')
301 time.sleep(7 if __debug__ else 2)
302 'turn off Van der Waals surface'
303 radiobutton_boxes[5].find_element_by_xpath('./input').click()
304 'animate vibrations'
305 vibration = self.webdriver.find_element_by_xpath(
306     '//input[contains(@name, "vibration_toggle")]')
307 logging.debug('vibration_checkbox: %s', vibration.get_attribute(
308     'outerHTML'))
309 vibration.click()
310 logging.warn('*** VERIFY VISUALLY vibration animated ***')
311 logging.warn('*** NOTE *** only works with certain files')
312 time.sleep(5 if __debug__ else 1)
313 vibration.click()
314 logging.warn('*** VERIFY VISUALLY vibration stopped ***')
315 time.sleep(5 if __debug__ else 1)
316 'verify text from database present on page'
317 data_div = self.webdriver.find_element_by_xpath(
318     '//div[contains(@class, "molecule_data")]')
319 logging.debug('data div contains: %s', data_div.text)
320 self.assertIn('Filename: methanol.cml', data_div.text)
321
322 ...
323 ~~~~
324 Don't search for space after "Task:" if no task was in the file.
325 HTML will eliminate the following space and the search will fail.
326 ...
327
328 self.assertIn('Task:', data_div.text)

```

Figure 5.18 (cont.) : Continuation of figure 5.18.

The view should have the ability to display a single molecule item view and a multiple molecule item view depending on the number of CML file(s) selected for viewing from the list view in accordance to the specification. This was implemented in Figure 5.19 as shown below:

```

57 def view(request, items=''):
58     """
59     Visual display of molecules using JSmol
60     """
61     selected = items.split('+')
62     logging.debug('selected: %s', repr(selected))
63     molecules = [Molecule.objects.get(id=s)
64                  if s.isdigit() else s for s in selected]
65     logging.debug('molecules: %s', repr(molecules))
66     if len(selected) > 1:
67         logging.debug('using multiview')
68         width = 100 / len(selected) - 5
69         div_width = width
70         side = min(35, width)
71         template = 'multiview'
72     else:
73         logging.debug('using singleview')
74         div_width = 96
75         side = 40
76         template = 'singleview'
77     return render(
78         request,
79         'viewer/view.html',
80         {
81             'selected': molecules,
82             'div_width': div_width,
83             'side': side,
84             'template': template,
85         }
86     )
87
88 def load(request, item):
89     """
90     Return data for a single molecule
91
92     This is a "helper" view, used by `view` indirectly, as the JavaScript
93     loaded by `view` then uses JSmol to load the molecule data from the
94     database, e.g.: `load load/0.cml`. This function is then called through
95     the URL dispatcher.
96
97     This is tested, therefore, by Views.test_display_view, and doesn't need
98     its own unit test.
99     """
100     if item.isdigit():
101         molecule = Molecule.objects.get(id=int(item))
102         return HttpResponse(molecule.data, content_type='text/plain')
103     else:
104         return redirect('/static/jsmol/data/%s' % item)

```

Figure 5.19: The implementation of item view.

The `view` function make use of a helper function called `load` to ensure the specifications were meet. The description of what the `load` function does is provided on Lines 90–98 in Figure 5.19.

The most interesting aspect of the implementation is the rendered `view.html`. It has a variety of languages ranging from HTML5, Django template language (explained in section 2.8), JavaScript and JSmol all working together in the single file as seen in Figure 5.20.

```

script
1
2 {% extends 'public/base.html' %}
3 {% load filters %}
4 {% block body_block %}
5
6 {% comment %}
7 The template construct {{item.id|default:item|attribute_safe}} appears a few
8 times below. It is necessary because we are allowing both Molecule objects
9 and regular filenames as `item`s. The filenames have no `id` attribute so
10 we use the filename instead; and the `attribute_safe` filter removes characters
11 that would make the name unsuitable for use as an HTML attribute.
12 {% endcomment %}
13
14 <script type="text/javascript" src="/static/jsmol/JSmol.min.js"></script>
15 <script type="text/javascript">
16     console.log("testing javascript is being executed");
17
18
19 // these next 3 lines serve to namespace the script.
20 // in order to avoid namespace pollution of the `window`.
21 if (typeof(com) == "undefined") com = {};
22 if (typeof(com.dayo) == "undefined") com.dayo = {};
23 com.dayo.jmol = {};
24
25 // see documentation on the info object here:
26 // http://wiki.jmol.org/index.php/Jmol_JavaScript_Object/Info
27 com.dayo.jmol.info = {
28     height: "100%",
29     width: "100%",
30     debug: false,
31     color: "white",
32     serverURL: "https://chemapps.stolaf.edu/jmol/jsmol/php/jsmol.php",
33     use: "HTML5",
34     j2sPath: "/static/jsmol/j2s",
35     disableJ2SLoadMonitor: false,
36     disableInitialConsole: true,
37     allowjavascript: true,
38     script: "set zoomlarge false; set antialiasdisplay;"

```

Figure 5.20: Item view HTML code fragment.

We have given detailed information about JSmol in Section 3.13. Therefore, we are just going to briefly describe how they all work together here. We initialize JSmol and set the options for the info object in Figure 5.20 . Then we wrote the JSmol script to execute from the data attribute. After which we created the applet and initialize it with JSmol scripting.

```

39  };
40
41  /* toggle code which is monkey-patched onto the radiobuttons and checkboxes
42     by the window.onload script. it fetches the JSmol code to execute from the
43     data attributes.
44     see https://developer.mozilla.org/
45        en/docs/Web/Guide/HTML/Using_data_attributes
46  */
47
48  com.dayo.jmol.toggle = function(event) {
49      console.log("toggle() called");
50      var cdj = com.dayo.jmol;
51      var element = event.target;
52      console.log("clicked: " + element);
53      var appletName = element.getAttribute("data-applet");
54      var applet = eval(appletName);
55      console.log("applet " + appletName + ": " + applet);
56      var appletdiv = eval(appletName + "div");
57      console.log("appletdiv: " + appletdiv.tagName);
58      var itemdiv = appletdiv.parentNode;
59      console.log("itemdiv: " + itemdiv.tagName);
60      var script = element.getAttribute(element.checked ?
61          "data-checked" : "data-unchecked");
62      var span = element.parentNode;
63      console.log("parent node: " + span.tagName);
64      var label = span.querySelector("label").textContent;
65      console.log("label: " + label);
66      console.log("running: " + script);
67      Jmol.script(applet, script);
68
69      // enable or disable transparency and mep_toggle depending on isosurface
70      console.log("input name: " + element.getAttribute("name"));
71      if (element.getAttribute("name") == "surface") {
72          if (label == "surface off") {
73              itemdiv.querySelector(
74                  "input[name='transparency']").disabled = true;

```

Figure 5.20 (cont.) : Continuation of item view HTML code fragment.

Finally, we created a div class for the controls and arranged each of them in another individual div class as seen in Figure 5.20 continuation.

```

75         itemdiv.querySelector(
76             "input[name='mep_toggle']").disabled = true;
77     } else {
78         itemdiv.querySelector(
79             "input[name='transparency']").disabled = false;
80         itemdiv.querySelector(
81             "input[name='mep_toggle']").disabled = false;
82     }
83 }
84 };
85
86 window.onload = function() {
87     var cdj = com.dayo.jmol;
88     console.log("window.onload() started");
89     var inputs = document.getElementsByTagName("input");
90     for (var i = 0; i < inputs.length; i++) inputs[i].onclick = cdj.toggle;
91     console.log("window.onload() finished");
92 };
93
94 </script>
95 <div class="container {{template}}">
96     <div class="header"><a href="/list">Home</a></div>
97     <!-- selected={{selected}} div_width={{div_width}} template={{template}} --
98     {% for item in selected %}
99     <form name="form_{{item.id|default:item|attribute_safe}}">
100     <div id="item_{{item.id|default:item|attribute_safe}}div" class="inline"
101         style="...">
102     <div id="applet_{{item.id|default:item|attribute_safe}}div" class="square"
103         style="...">
104     <script type="text/javascript">
105         console.log("creating applet for item {{item}}");
106         (function() {
107
108             // this both creates the applet and initializes it with Jmol scripting.
109             var cdj = com.dayo.jmol;
110             var appletId = "applet_{{item.id|default:item|attribute_safe}}";

```

Figure 5.20 (cont.) : Continuation of item view HTML code fragment.

```

111     Jmol.getApplet(appletId, cdj.info); // creates variable of same name
112     Jmol.script(eval(appletId),
113       "load /load/{{item.id|default:item}}; cdj_surface = '';" +
114       " cdj_mapping = 'none'; cdj_transparent = false; sync . on;" +
115       " if ({atomno < 10}.partialcharge == 0) {calculate partialcharge};");
116   }}());
117
118 </script>
119 </div>
120
121 <div class="controls">
122   <h3>Controls:</h3>
123   <div class="grouping">
124     <span class="labeled_input">
125       <input type="radio" name="appearance" checked
126         data-applet="applet{{item.id|default:item|attribute_safe}}"
127         data-checked="select *; cartoons off; spacefill 23%; wireframe 0.15;"/>
128       <label for="appearance">ball and stick</label>
129     </span>
130   </div>
131
132   <div class="grouping">
133     <span class="labeled_input">
134       <input type="radio" name="appearance"
135         data-applet="applet{{item.id|default:item|attribute_safe}}"
136         data-checked="select *; cartoons off; wireframe -0.1;"/>
137       <label for="appearance">wireframe</label>
138     </span>
139   </div>
140
141   <div class="grouping">
142     <span class="labeled_input">
143       <input type="radio" name="appearance"
144         data-applet="applet{{item.id|default:item|attribute_safe}}"

```

Figure 5.20 (cont.) : Continuation of item view HTML code fragment.

```

257     {%endif %}
258 </div><!-- controls -->
259 <div class="molecule_data">
260     <h3>Molecule information</h3>
261     <ul>
262         <li>Filename: {{item.source|default:item}}</li>
263         <li>Task: {{item.task|default:''}}</li>
264         <li>Method: {{item.method|default:''}}</li>
265         <li>Basis Set Label: {{item.basis_set_label|default:''}}</li>
266         <li>SMILES: {{item.smiles|default:''}}</li>
267         <li>InChI: {{item.inchi|default:''}}</li>
268     </ul>
269 </div><!-- molecule_data -->
270 </div><!-- item -->
271 </form>
272 {% endfor %}
273 </div><!-- container -->
274 <!-- endblock -->
275 {% endblock %}
276 {% comment %}
277 vim: tabstop=8 expandtab shiftwidth=4 softtabstop=4
278 {% endcomment %}

```

Figure 5.20 (cont.) : Continuation of item view HTML code fragment.

5.9.1 Item View

Single View

Running our application, selecting an uploaded CML file(s) (molecule) from the list view, and clicking on the view button at the bottom of the list view page takes us to the single or multiple molecule item view where a user can visualize the CML file (molecule) in different visualization models as seen in the Figures 5.21-5.34.

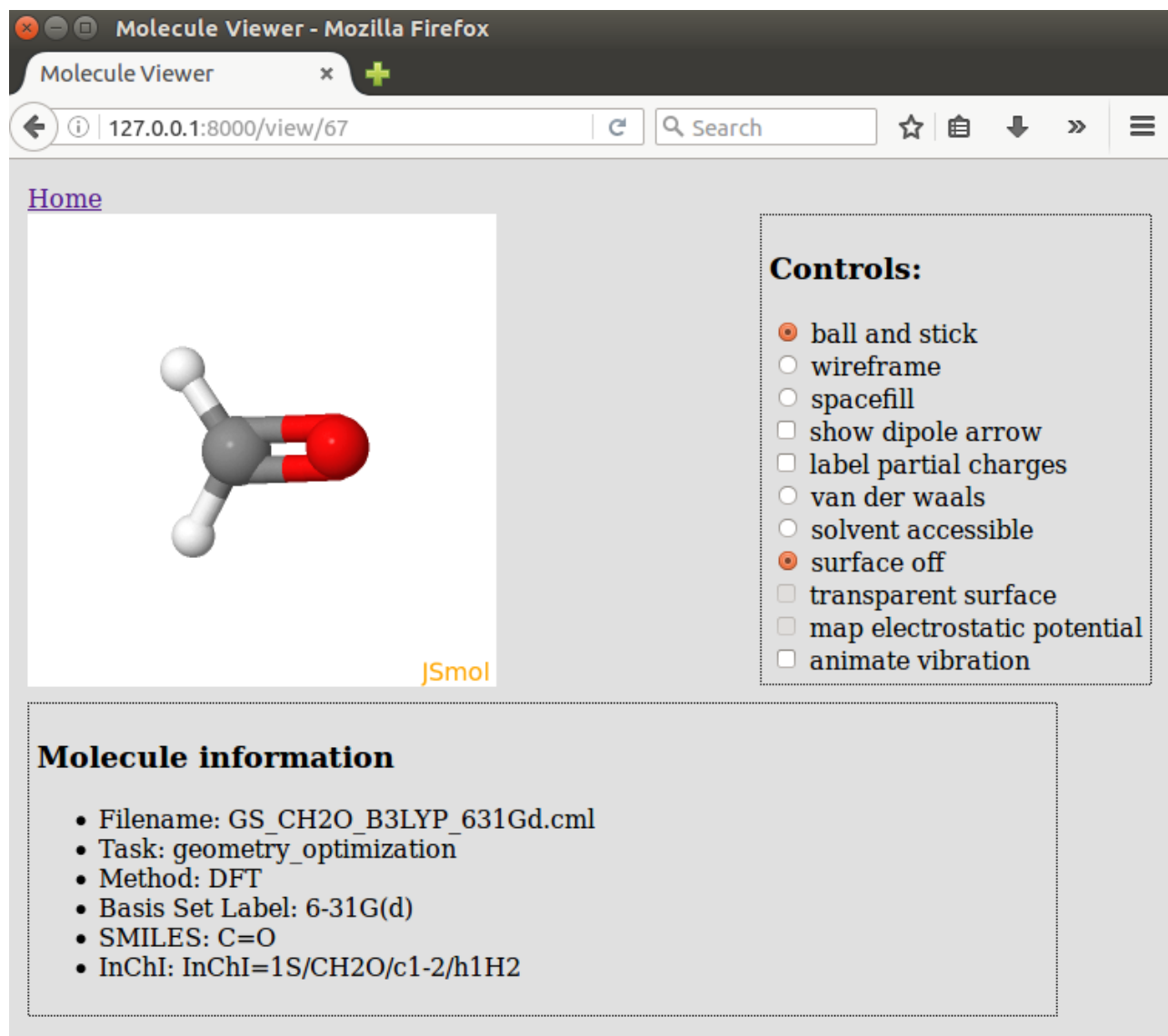


Figure 5.21: The visualization of an uploaded molecule in balls and stick model.

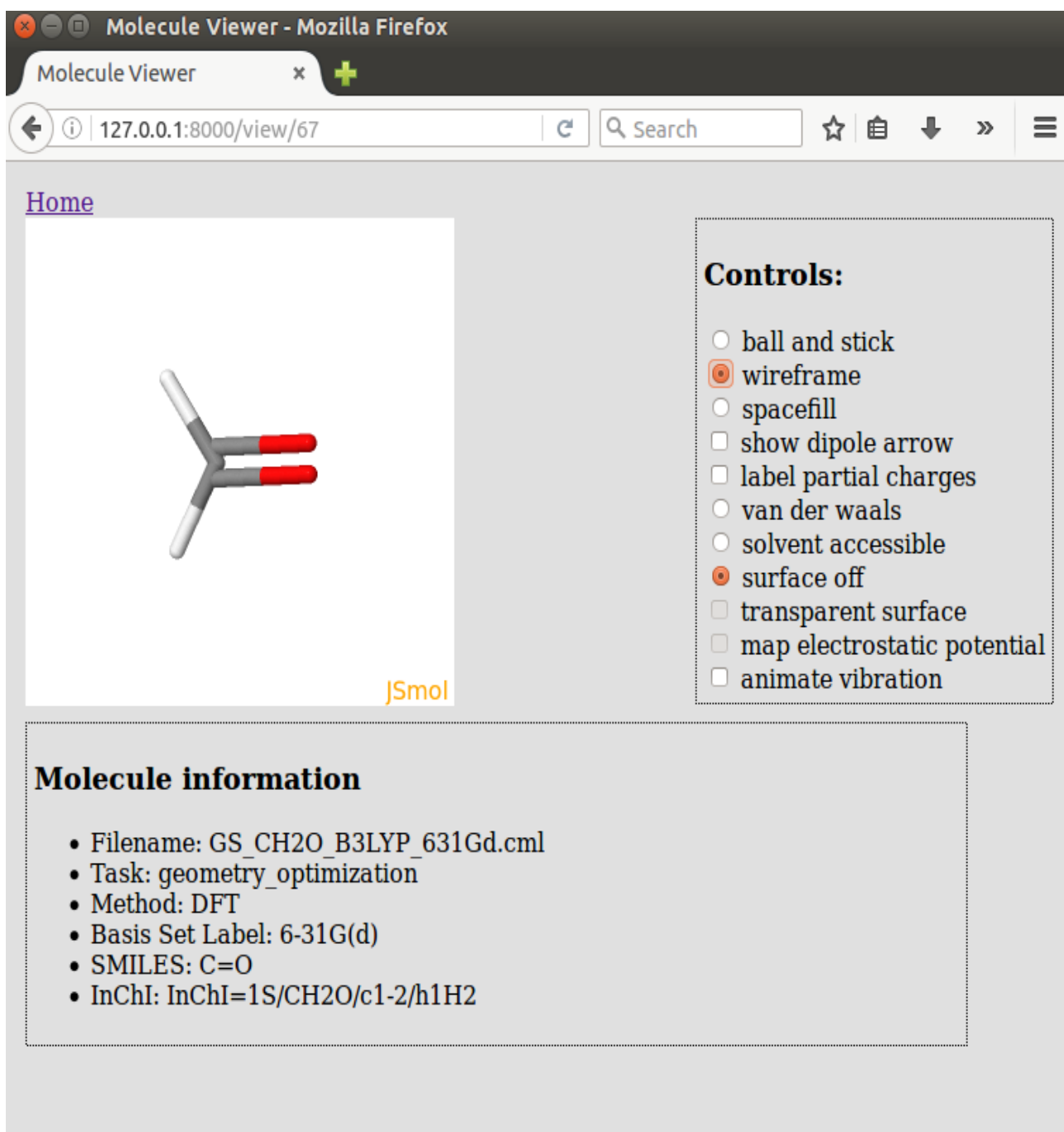


Figure 5.22: The visualization of an uploaded molecule in wireframe model.

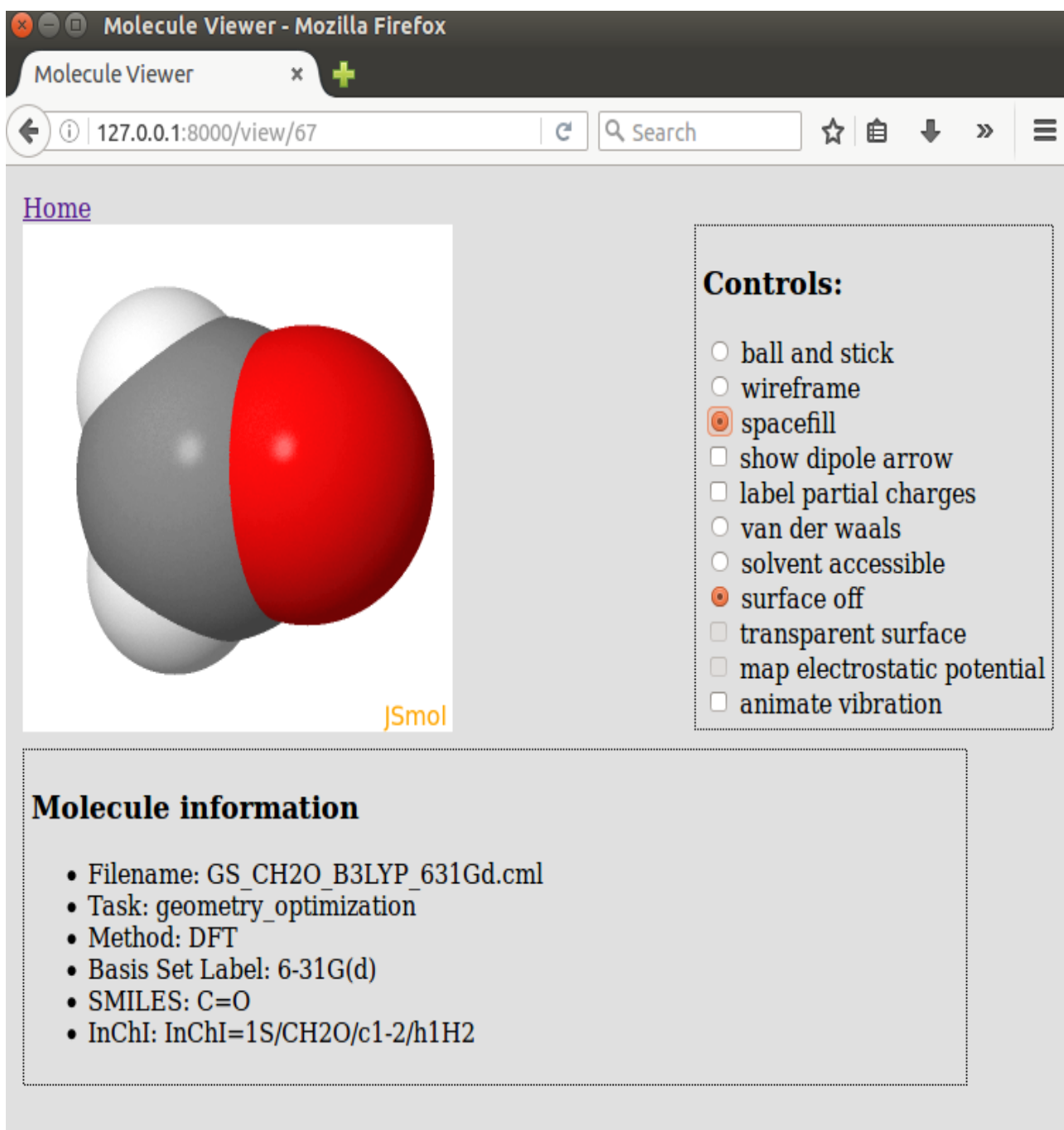


Figure 5.23: The visualization of an uploaded molecule in spacefill model

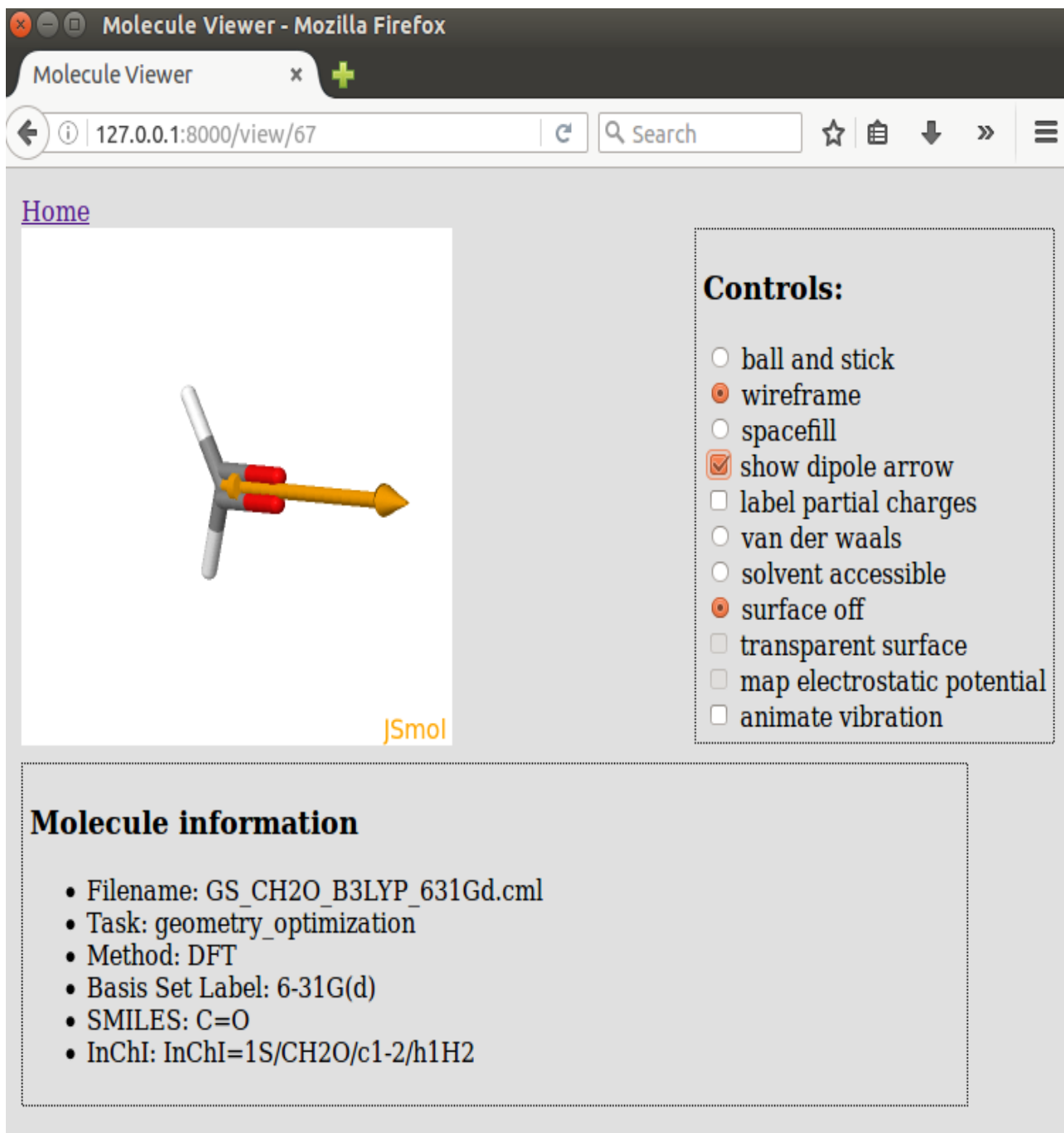


Figure 5.24: The visualization of dipole arrow.

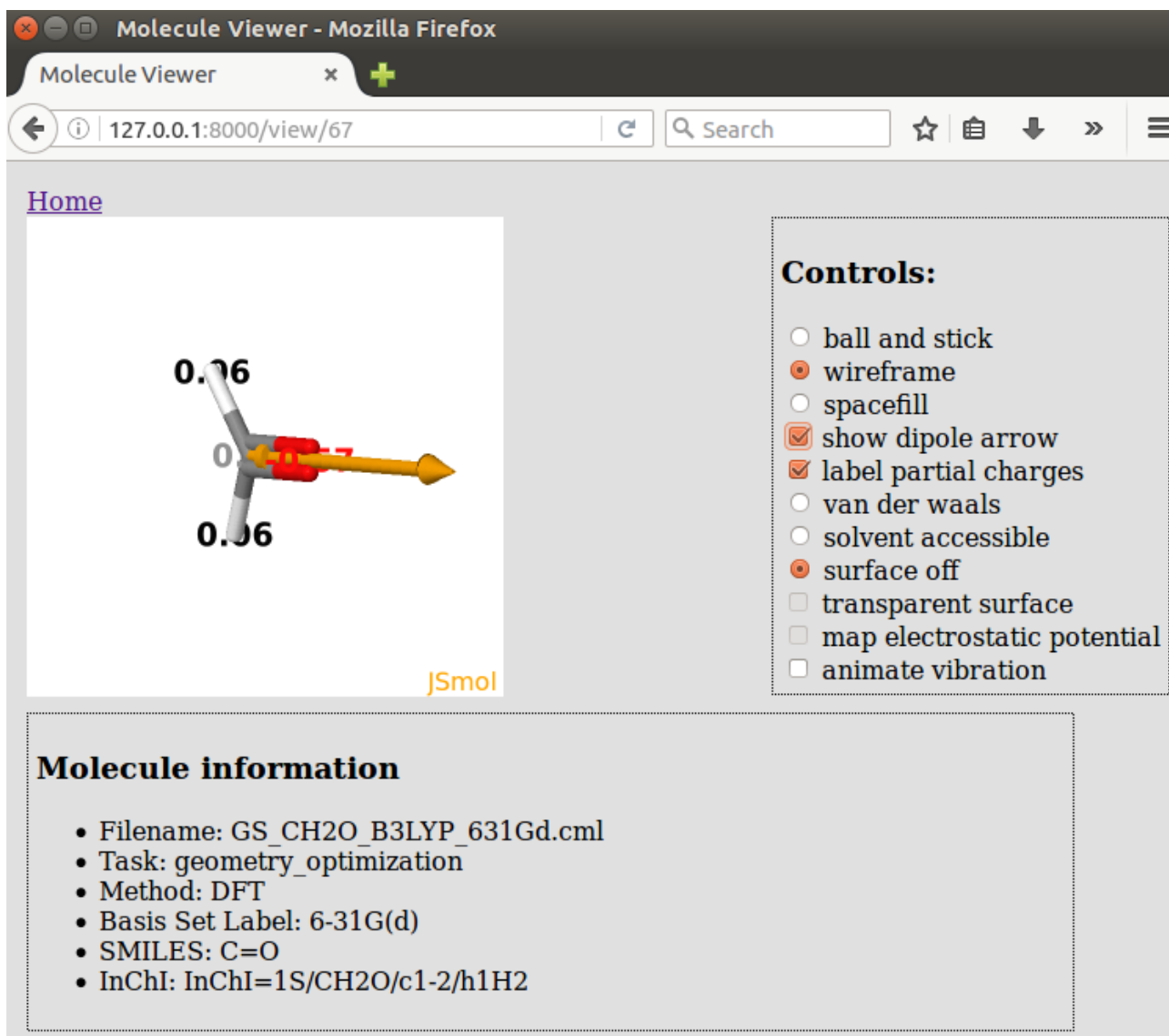


Figure 5.25: The labeling of partial charges and dipole arrow.

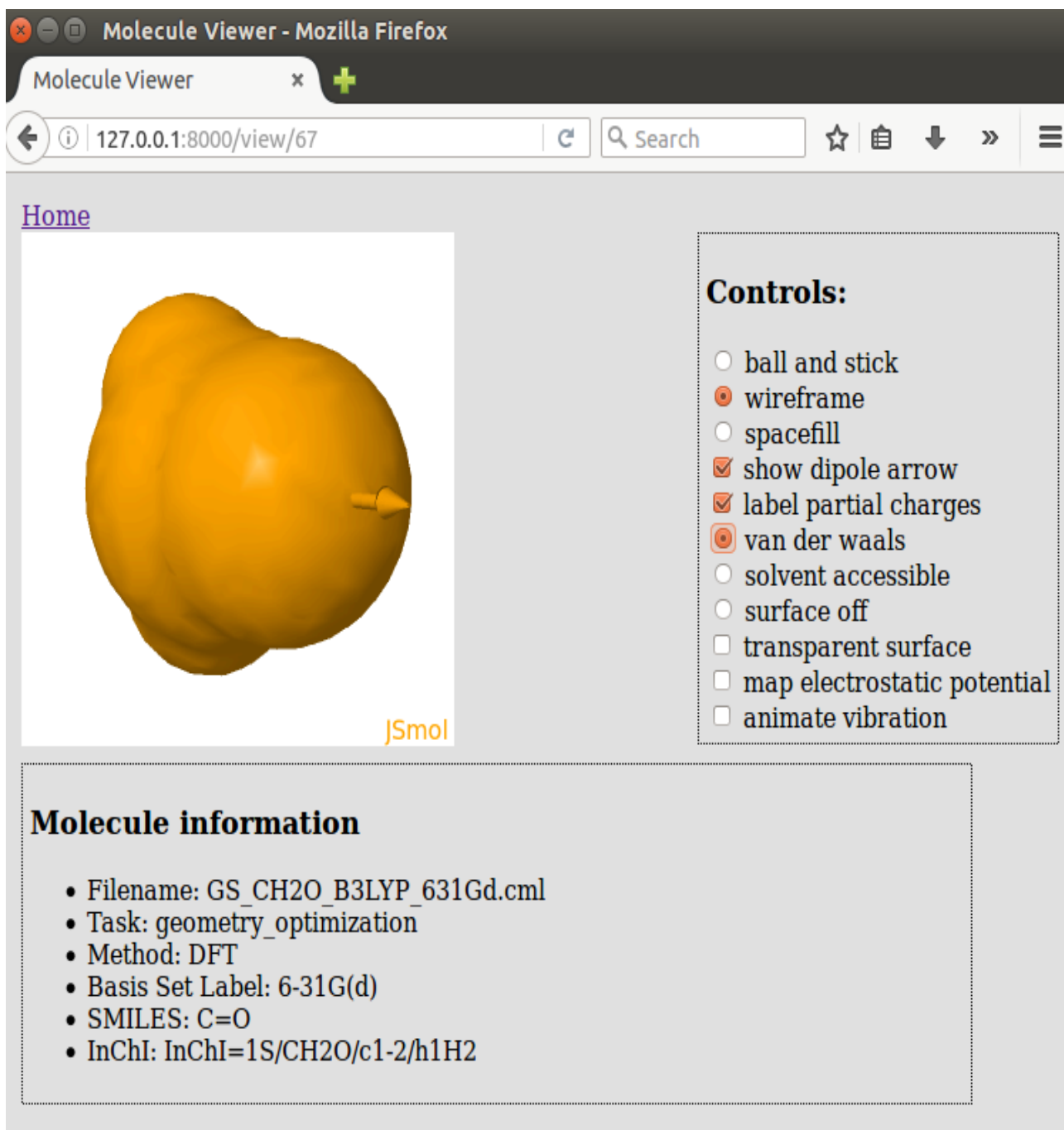


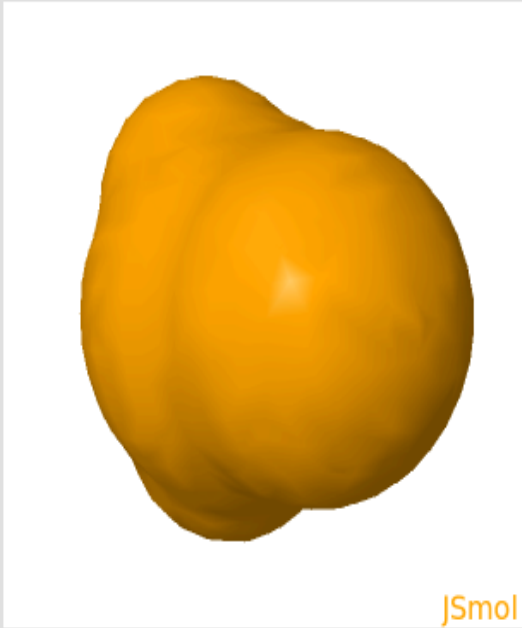
Figure 5.26: The visualization of Van-der waal surface and dipole arrow.

Molecule Viewer - Mozilla Firefox

Molecule Viewer x +

127.0.0.1:8000/view/67 Search

[Home](#)



JSmol

Controls:

- ☐ ball and stick
- ☒ wireframe
- ☐ spacefill
- ☐ show dipole arrow
- ☐ label partial charges
- ☐ van der waals
- ☒ solvent accessible
- ☐ surface off
- ☐ transparent surface
- ☐ map electrostatic potential
- ☐ animate vibration

Molecule information

- Filename: GS_CH2O_B3LYP_631Gd.cml
- Task: geometry_optimization
- Method: DFT
- Basis Set Label: 6-31G(d)
- SMILES: C=O
- InChI: InChI=1S/CH2O/c1-2/h1H2

Figure 5.27: The visualization of solvent accessible surface.

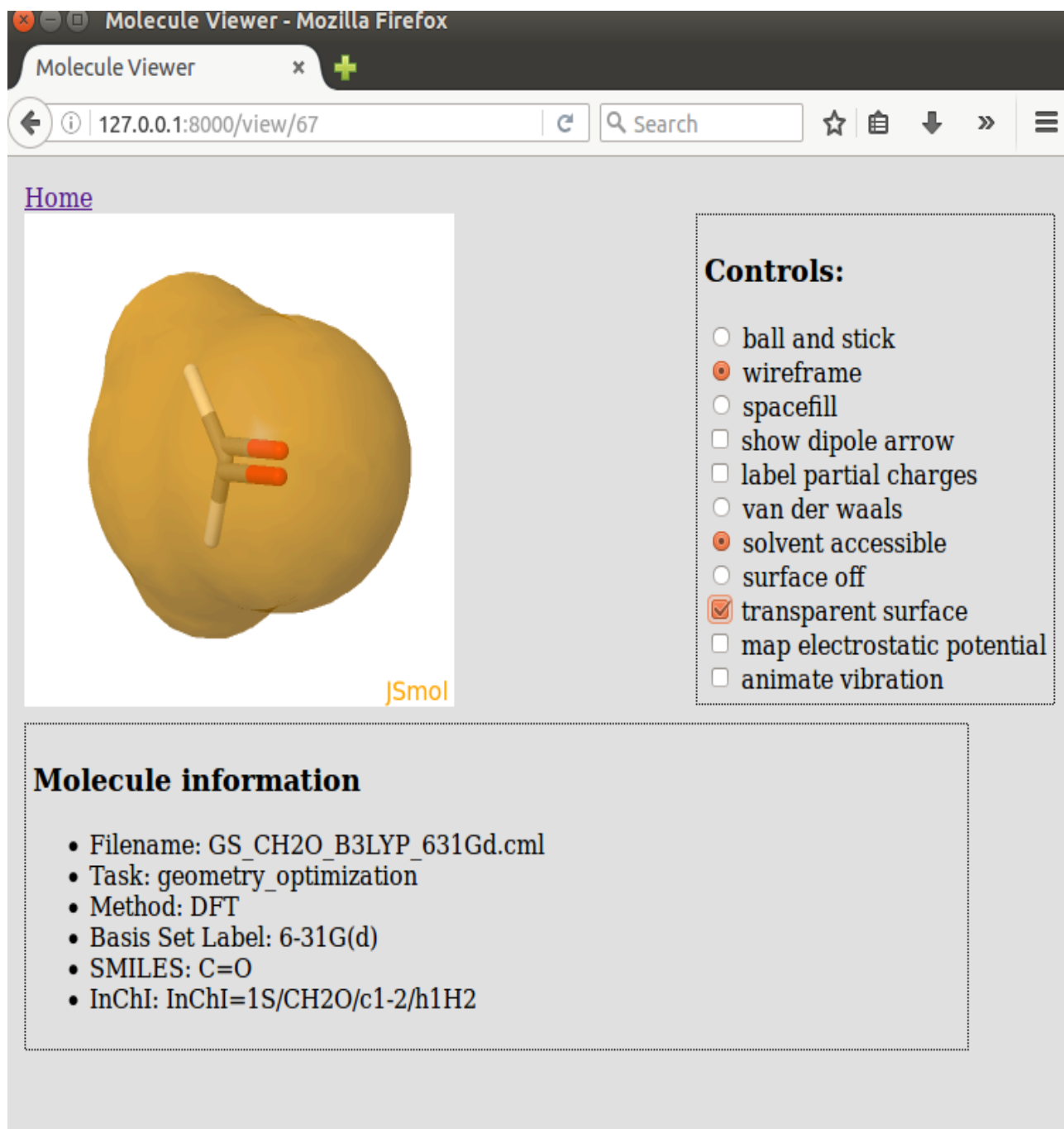


Figure 5.28: The visualization of transparent surface.

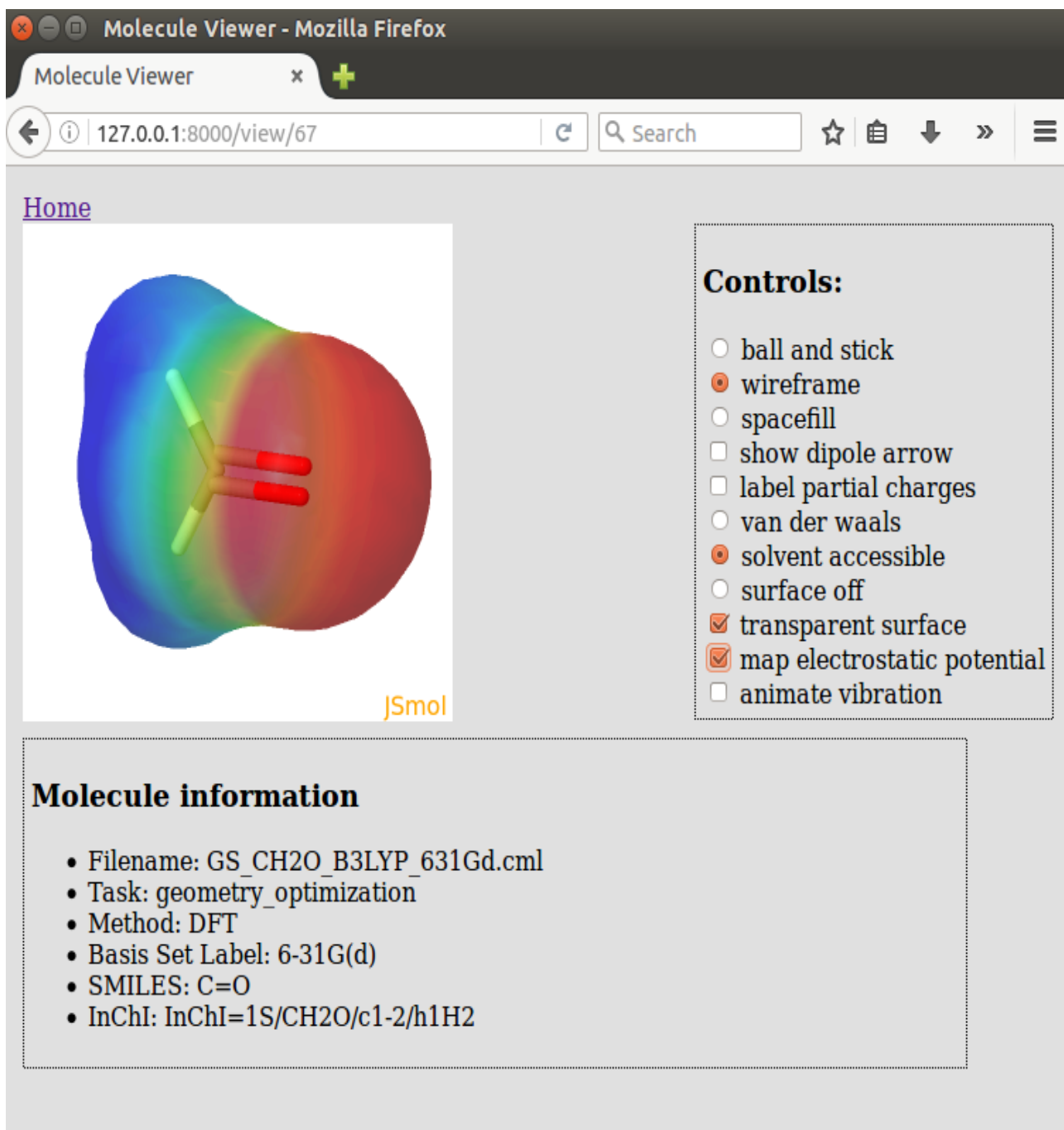


Figure 5.29: The mapping of electrostatic potential.

Multiple View

As part of the requirements, selecting multiple CML files from the list view, and clicking on the view button at the bottom of the list view page takes us to the multiple molecule item view where a user can visualize and compare two or three molecules in different visualization models as seen in Figures 31-34.

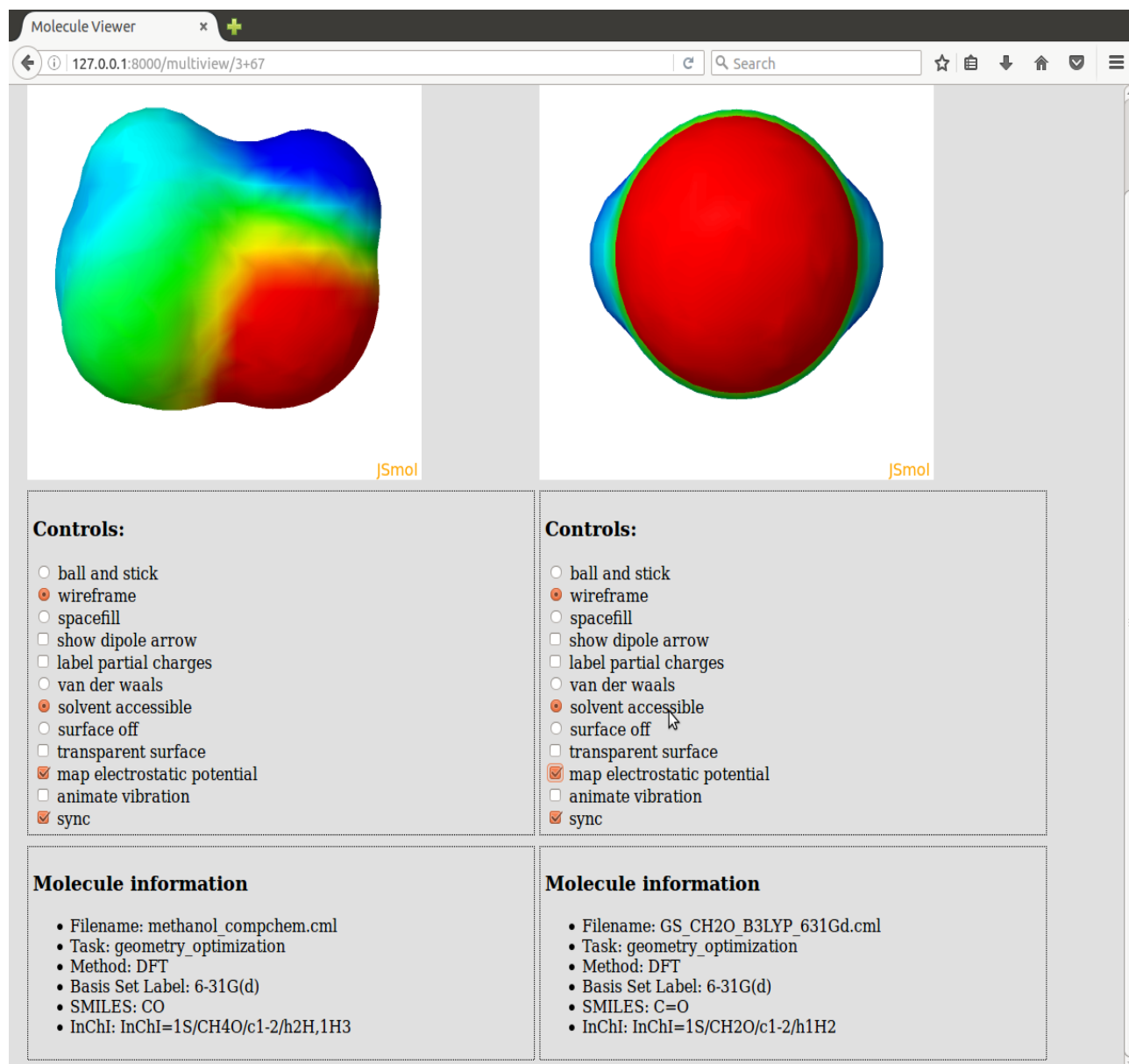


Figure 5.30: The visualization of multiple molecule item view electrostatic potential.

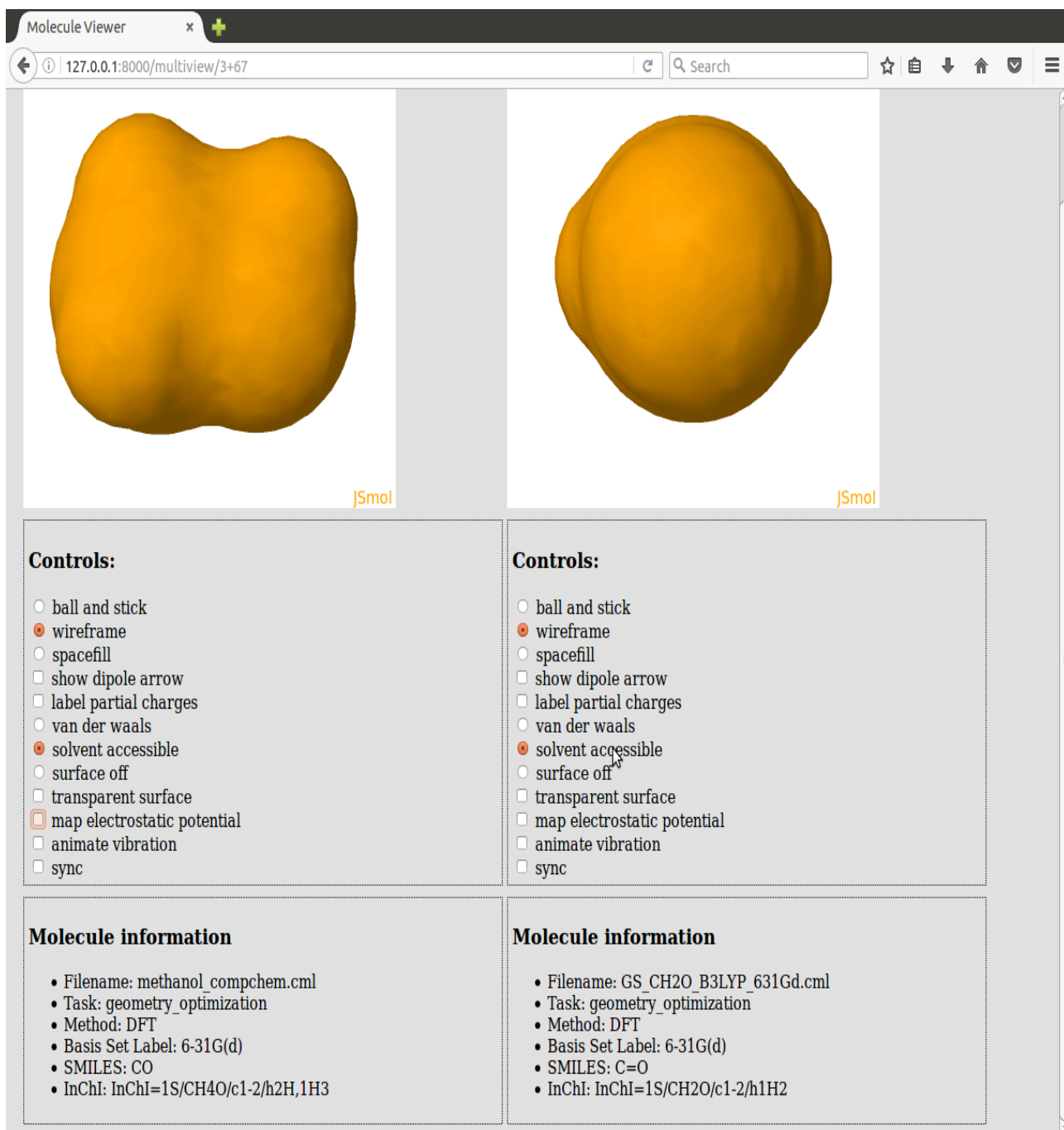


Figure 5.31: The visualization of multiple molecule item view solvent accessible surface.

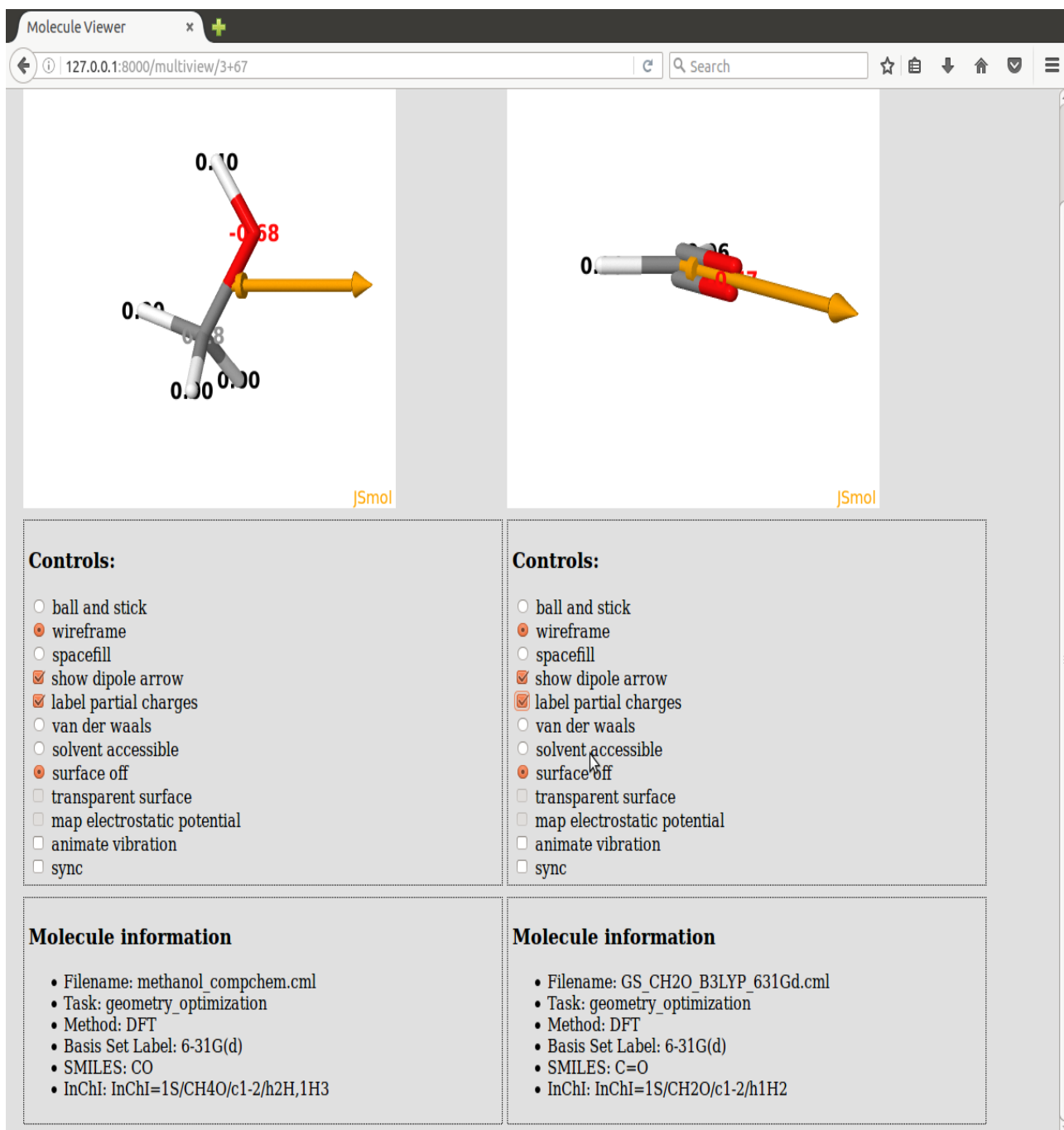


Figure 5.32: The visualization of multiple molecule item view partial charge.

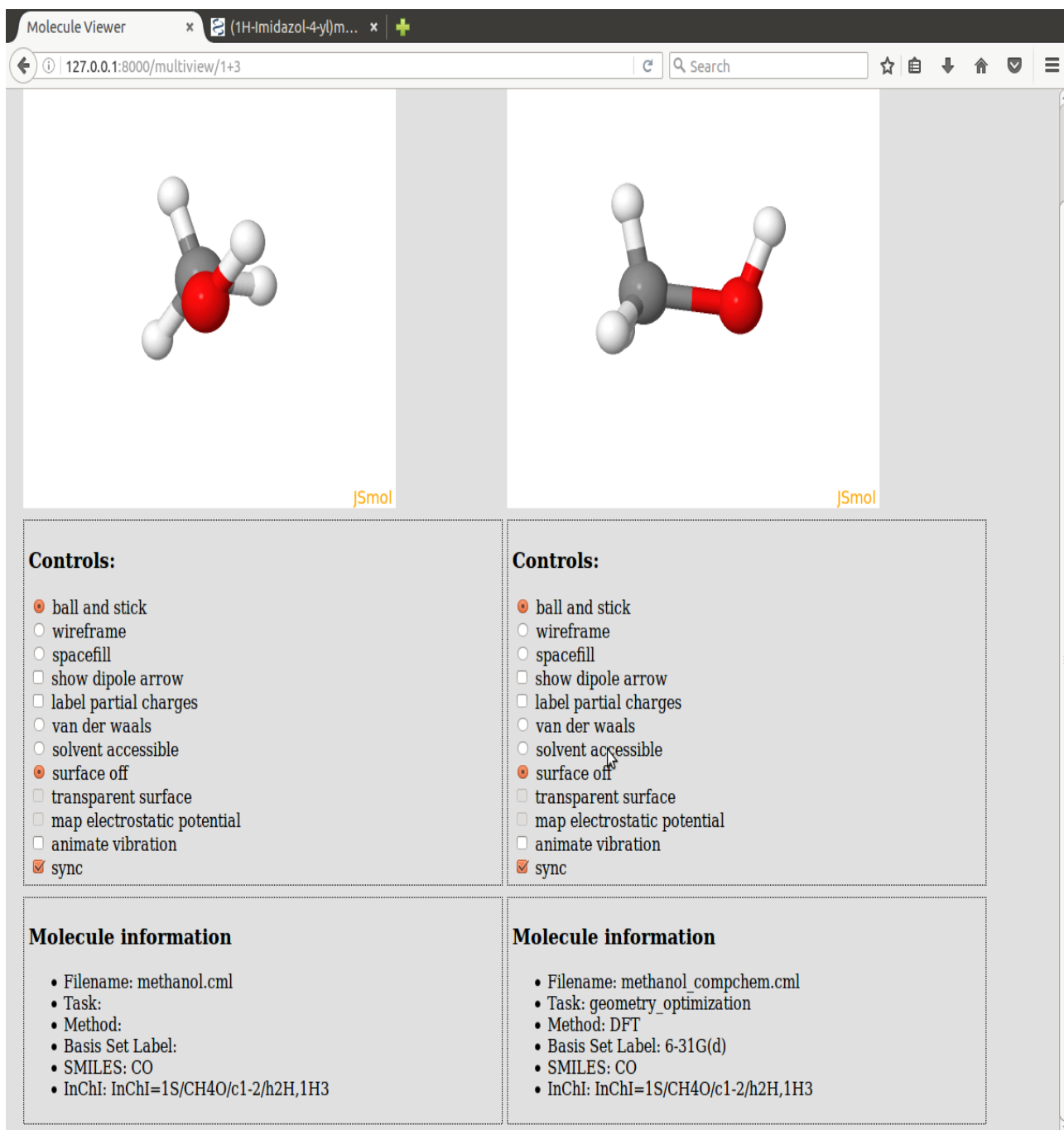


Figure 5.33: The visualization of multiple molecule item view ball and stick.

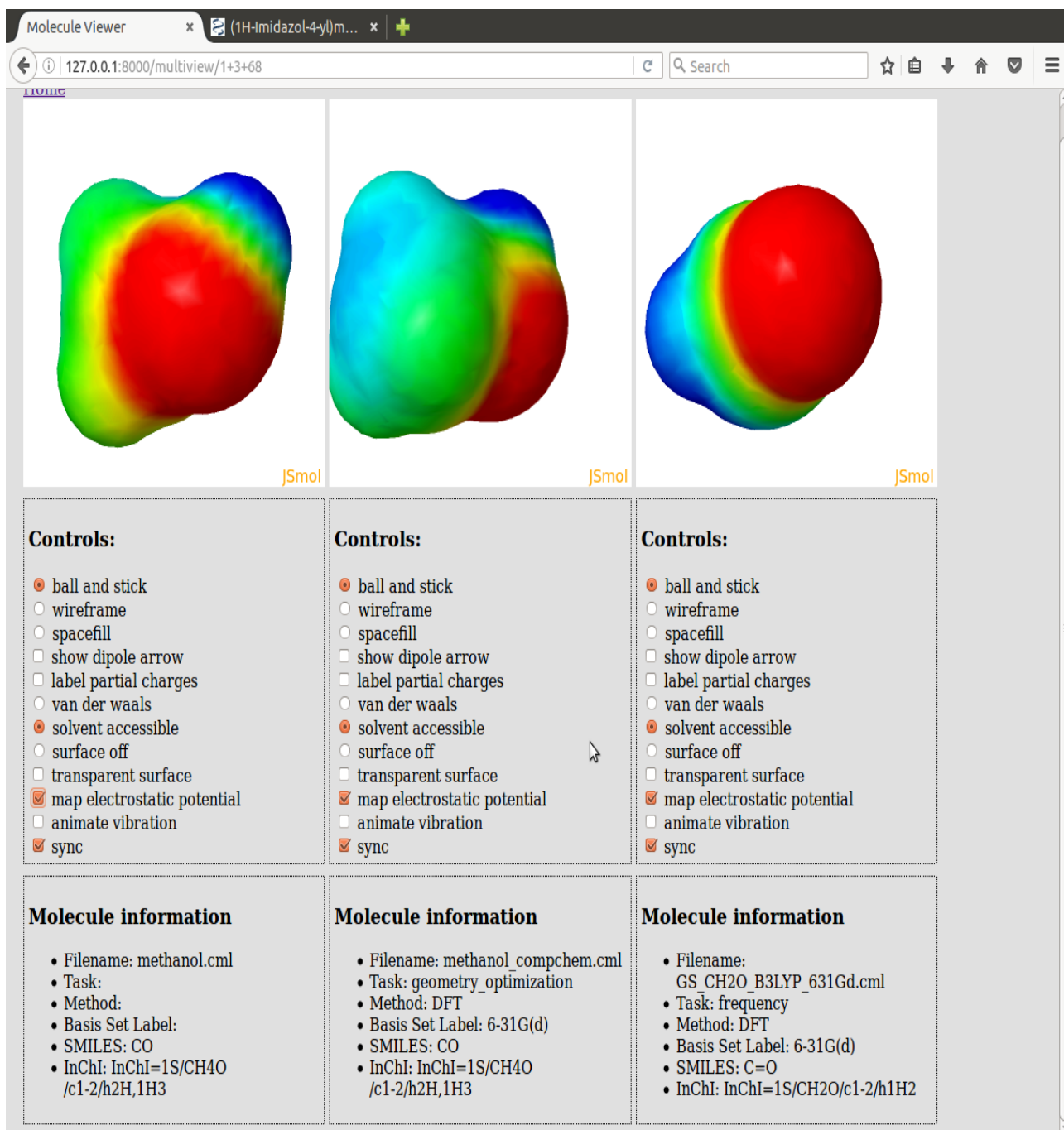


Figure 5.34: The visualization of multiple molecule item view (3 Molecules) ESP.

5.10 Implementing Search Functionality

We implemented a search functionality for the list view. Search can be implemented separately, but we made it part of the list view since it has a strong association with it. Figure 5.35 shows the test case for the search functionality. The implementation code is part of the list view implementation code.

```
335 def test_search_function(self):
336     """
337     Optional Search functionality is part of Listing
338
339     Search can be:
340     | a single word or part of a word: energy
341     | a fieldname followed by a word or part: source:ch4
342     | Multiple words or 'fieldname:value' pairs will be ANDed
343     """
344
345
346     'first make sure there is something in the database'
347     logging.debug('starting test_search_function')
348     self.upload_molecule_file('methanol_compchem.cml')
349     self.webdriver.open('/')
350     inputfield = self.webdriver.find_element_by_id('searchbox')
351     button = self.webdriver.find_element_by_id('search')
352     inputfield.send_keys('geometry')
353     logging.debug('looking for slowdown here: 0')
354     button.click()
355     logging.debug('looking for slowdown here: 1')
356     time.sleep(0.5)
357     logging.debug('looking for slowdown here: 2')
358     self.assertNotEqual(self.webdriver.find_elements_by_id('//tr'),
359                         Molecule.objects.count() + 1) # add 1 for header
360     logging.debug('looking for slowdown here: 3')
361
```

Figure 5.35: The implementation of search functionality.

5.11 Secondary Requirement Implementation

As described in Section 4.7, they are basically modifications to the design and addition of some new requirements (see Section 4.7). The secondary requirement includes proper arrangement of the control buttons, addition of sync toggle for multiple molecule item view and the implementation of RESTful API for the application as seen in the docstring of figure 5.36.

```
380 def test_spec2(self):
381     """
382     from specification 2 document
383
384     In both single- and multi-view:
385
386     * Make JSmol applets square (assign width and height for the <div> in css)
387     * Place Jmol controls in <div> element and arrange them "nicer".
388     * Add link taking the user back to list view at the top right corner.
389     * Add control for transparency of surfaces
390
391     * For single molecule view:
392
393     * Place controls in a floating <div> in the right side of the page:
394
395     CSS: div.controls{ width: 500px; float: right; }
396
397     * Have single molecule-view under http://{server:port}/view
398     * Use RESTful interface for single molecule view:
399
400     http://{server:port}/view/{mol_identfier}
401     E.g. http://{server:port}/view/6 loads molecule with ID 6
402     """
403     logging.debug('starting test_spec2')
404     self.webdriver.open('/view/c6h6.smol')
405     canvas = self.webdriver.find_element_by_tag_name('canvas')
406     self.assertEqual(canvas.size['width'], canvas.size['height'])
407     if Molecule.objects.count() < 2:
408         self.upload_molecule_file('methanol_compchem.cml')
409     logging.info('count of Molecules: %d', Molecule.objects.count())
410     check = self.webdriver.find_element_by_xpath('//tr[2]/td[1]/input')
411     check.click()
412     check = self.webdriver.find_element_by_xpath('//tr[3]/td[1]/input')
413     check.click()
414     self.webdriver.find_element_by_name('display').click()
415     self.webdriver.wait_for_page_load()
416
417     ...
418
419     * For Multiview:
420
421     * Have multi molecule-view under http://{server:port}/multiview
422     ...
423
424     RESTFUL API for multiview e.g
425     http://{server:port}/multiview/2+6+10
426     compares (database Ids 2,6 and 10)
427     ...
428
429     self.assertTrue(bool(re.match(r'/multiview/\d+[\+]\d+$',
430         urlparse.urlsplit(self.webdriver.current_url).path)))
431
432     ...
433
```

Figure 5.36: The test written for the secondary requirement.


```

434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490

```

```

    • Add control to toggle sync of JSmol applets
    (Jmol command "sync on"). Sync can be on by
    default.
    See: https://chemapps.stolaf.edu/jmol/docs/index.htm#sync
    ...

self.assertTrue(bool(
    self.webdriver.find_element_by_name('sync_toggle')))
canvas = self.webdriver.find_element_by_tag_name('canvas')
self.assertEqual(canvas.size['width'], canvas.size['height'])

'Place Jmol controls in <div> element and arrange them "nicer".'
self.assertEqual(
    self.webdriver.find_element_by_tag_name(
        'input').find_element_by_xpath(
            '../div[2]').get_attribute('class'), 'controls')
self.assertEqual(
    self.webdriver.find_element_by_xpath('//a[@href="/list"]').text,
    'Home'
)
checkbox_labels = self.webdriver.find_elements_by_xpath(
    '//input[@type="checkbox"]/../label')
logging.debug('checkbox_labels: %s', repr(checkbox_labels))
self.assertIn('transparent surface', [c.text for c in checkbox_labels])

...
~~~~~
    • Have single molecule-view under http://{server:port}/view
    • Use RESTful interface for single molecule view:

http://{server:port}/view/{mol_identifier}
E.g. http://{server:port}/view/6 loads molecule with ID 6
...

self.webdriver.find_element_by_xpath('//a[@href="/list"]').click()
items = self.webdriver.find_elements_by_tag_name('input')
items[5].click()
self.webdriver.find_element_by_name('display').click()
self.webdriver.wait_for_page_load()
logging.info('current_url: %s', self.webdriver.current_url)
self.assertTrue(bool(re.match(r'/view/\d+$',
    urlparse.urlsplit(self.webdriver.current_url).path)))

...
~~~~~
In single view, add box to display textual information from database.
...

infobox = self.webdriver.find_element_by_class_name('molecule_data')
self.assertEqual(infobox.tag_name, 'div')

...
Extract more textual information from CHL/CompChem file,
basically Method, SMILES, InChI... and store in the database
...

self.assertTrue(bool(Molecule.objects.first().smiles))

```

Figure 5.36 (cont.) : Continuation of figure 5.36.

5.12 Testing Milestones

In order to ensure that we have covered all milestones in Section 4.5, we wrote a test case as seen in Figure 5.37.

```
493 class Milestones(SeleniumTestCase):
494     """
495     Test milestones as specified in spec1 and spec2 doc
496     """
497     def test_milestones(self):
498         """
499         Test to ensure milestone is reached
500
501         (1) Django webapp with item view showing a single molecule
502             with JSmol (e.g a file loaded from /static/
503         (2) Add basic controls for visualization (Balls & Sticks; wireframe;
504             spacefill).
505         (3) Ability to upload a "small" CML file, store it (as string?)
506             in the DB, add entry to list view, item view loads molecule
507             structure from DB (instead of /static)
508             [*** already tested in test_upload_view above***]
509         (4) Upload "large" CML/CompChem file (instead of "small") and
510             - while handling the uploaded data - extract the CML molecule
511             (effectively the content of the "small" CML file) from the
512             larger XML tree. Only store the small CML molecule(s).
513             [*** already tested in test_upload_view above***]
514         (5) Also extract cml:task and store along with CML molecule
515             in DB and display in list and item view.
516         (6) Extract dipoleMomentVector (three floats), store in DB and
517             use it to draw arrow in JSmol
518         (7) Extract netAtomicCharge (one float per atom), store in DB and
519             load this data as partial charges into JSmol molecule,
520             add ESP surface
521         (8) Extract more textual data from CML:
522             Method
523             Basis set label
524             SMILES
525             InChI....
526         (9) Add animation for vibrations (only available when `task`
527             is `frequency_calculation`; for this we still need to figure
528             out how to load the required extra information into Jmol.
529         """
530
531         logging.debug('starting test_milestones')
532         # First milestone, static file
533         self.webdriver.open('/view/c6h6.smol')
534         page_text = self.webdriver.find_element_by_xpath('//html').text
535         logging.debug('page text: %s', page_text)
536         self.assertIn('Filename: c6h6.smol', page_text)
537         time.sleep(3 if __debug__ else 2)
538         # Milestone 2 already added using Views testing
539         # Milestones 3 and 4 already tested in test_upload_view
540         # Milestone 5 already tested in test_listing_view (list) and
541         # in test_display_view (item view).
542         # Milestone 6,7,8 all tested.
543         # Milestone 9
544         animate = self.webdriver.find_element_by_xpath(
545             '//input[contains(@name, "vibration_toggle")]')
546         animate.click()
547         time.sleep(5 if __debug__ else 2)
548         animate.click();
```

Figure 5.37: The test written to ensure total coverage of milestone.

5.13 Implementing Database

We implemented and tested our model which Django documentation describes as the single, definitive source of information about one's data as shown in figure 5.38 [12]. To create our model, we created a class that sub-classed `django.db.models.Model`. The specified fields of the model (database) in Section 4.3 are name, data, source, index, etc. as shown in Figure 5.39.

```
553
554 class DatabaseFields(TestCase):
555     ...
556     Extract database fields from CML files
557
558     Required:
559         Filename: (string)
560         Small CML molecule (<molecule> element)
561         Dipole Moment Vector (3 floats) [unneeded]
562         Net Atomic (partial) Charges (one float per atom) [unneeded]
563         Task (e.g. geometry optimization, frequency) (string)
564         Basis Set Label (string) [later]
565         Method (string) [later]
566         SMILES (string) [later]
567         InChI (string) [later]
568         ...
569         Program Name
570         Program Version
571         ... maybe more...
572     ...
573
574
575 def test_molecule_extraction(self):
576     logging.debug('starting test_molecule_extraction')
577     results = {'task': [], 'program': [], 'program_version': []}
578     logging.debug(os.listdir(os.path.dirname(TESTFILE)))
579     data = convert('viewer', [TESTFILE])
580     self.assertNotEqual(len(data), 0) # tests Filename and CML molecule
581
582     for molecule in data:
583         for field in results.keys():
584             if molecule['fields'][field]:
585                 results[field].append(molecule['fields'][field])
586
587     for field in results.keys():
588         logging.debug('checking len(results["%s"]) > 0', field)
589         self.assertNotEqual(len(results[field]), 0)
590     # new spec in spec2
591     self.assertNotIn('/', data[0]['fields']['source'])
592
```

Figure 5.38: The test written for the database.

```

1  from django.db import models
2  MB = 1024 * 1024
3
4  class Molecule(models.Model):
5      """
6      Representation of molecule for viewing with Jmol
7      """
8      name = models.CharField(max_length=128)
9      data = models.CharField(max_length=10 * MB)
10     source = models.CharField(max_length=1024)
11     index = models.IntegerField()
12     task = models.CharField(max_length=1024, default='')
13     program = models.CharField(max_length=1024, default='')
14     program_version = models.CharField(max_length=1024, default='')
15     basis_set_label = models.CharField(max_length=1024, default='')
16     method = models.CharField(max_length=1024, default='')
17     smiles = models.CharField(max_length=1024, default='')
18     inchi = models.CharField(max_length=1024, default='')
19
20     def __str__(self):
21         return '%s[%d]' % (self.source, self.index)

```

Figure 5.39: The application database (Models).

5.14 Data Extraction

Data extraction was implemented with lxml. The lxml XML toolkit is a Pythonic binding for the C libraries libxml2 and libxslt. It integrates the swiftness and XML capabilities of libxml2 and libxslt with the simplicity of Python API [49].

It is usually compatible but superior to the well-known ElementTree functionality-wise [49]. With lxml and other useful libraries, we wrote a code responsible for the conversion of CML files to fixture. The fixture are then used to provide test data to Django [11] as shown in the figure 5.40.

```

1  #!/usr/bin/python3
2  '''
3  convert cml files to `fixture` for providing initial data to django
4
5  see https://docs.djangoproject.com/en/1.10/howto/initial-data/
6  '''
7
8  import sys, os, json, logging, re
9  from lxml import etree
10 logging.basicConfig(level=logging.DEBUG if __debug__ else logging.INFO)
11
12 def convert(appname, filenames):
13     '''
14     convert all given files to list of dicts, which can be JSONified
15     filenames can actually be any file-type objects...
16     however a file-type object may not work with os.path.basename, so
17     it is cast to a string below.
18     '''
19     output = []
20     namespaces = {
21         'cml': 'http://www.xml-cml.org/schema',
22         'mods': 'http://www.loc.gov/mods/v3',
23     }
24     default = property(lambda: '')
25     for filename in filenames:
26         logging.debug('processing "%s"', filename)
27         index = 0
28         tree = etree.parse(filename)
29         molecules = tree.xpath('//*[local-name() = "molecule"]')
30         for molecule in molecules:
31             output.append({
32                 'model': '%s.molecule' % appname,
33                 'fields': None,
34             })
35             logging.debug(
36                 'extracting molecule "%s" at index %d',
37                 molecule.get('id'),
38                 index)
39             output[-1]['fields'] = {
40                 'name': molecule.get('id', ''),
41                 'data': etree.tostring(molecule).decode('utf8'),

```

Figure 5.40: The implementation of extraction.

```

42         'source': os.path.basename(str(filename)),
43         'index': index,
44     }
45     append_parameter_value(output[-1], molecule, 'task')
46     append_parameter_value(output[-1], molecule, 'program')
47     append_parameter_value(output[-1], molecule, 'programVersion')
48     append_parameter_value(output[-1], molecule, 'basisSetLabel')
49     append_parameter_value(output[-1], molecule, 'method')
50     append_parameter_value(output[-1], molecule, 'inchi')
51     append_parameter_value(output[-1], molecule, 'smiles')
52     index += 1
53     return output
54
55
56
57 def parameter(element, string, default=''):
58     """
59     Return the value of a parameter from the given element, or default
60
61     >>> element = etree.fromstring(
62     ...     '<c><b><a><parameter dictRef="cc:programVersion">\n'
63     ...     '    <scalar dataType="xsd:string">AM64L-G09RevD.01</scalar>\n'
64     ...     '</parameter></a></b></c>\n')
65     >>> logging.debug('element: %s', element)
66     >>> parameter(element, 'programVersion')
67     'AM64L-G09RevD.01'
68
69     >>> element = etree.fromstring(
70     ...     '<c><b><a><property dictRef="id:smiles" xmlns:id="/">\n'
71     ...     '    <scalar dataType="xsd:string">C(Cl)(Cl)Cl</scalar>\n'
72     ...     '</property></a></b></c>\n')
73     >>> logging.debug('element: %s', element)
74     >>> parameter(element, 'smiles')

```

Figure 5.40 (cont.) : Continuation of figure 5.40.


```

75         'C(Cl)(Cl)Cl'
76
77     >>> element = etree.fromstring(
78         ...     '<c><b><a><identifier convention="iupac:inchi"'
79         ...     ' value="InChI=1S/CHCl3/c2-1(3)4/h1H"/></a></b></c>\\n')
80     >>> logging.debug('element: %s', element)
81     >>> parameter(element, 'inchi')
82     'InChI=1S/CHCl3/c2-1(3)4/h1H'
83
84     ...
85     value = default
86     here = element
87     while here != None:
88         logging.debug('searching for parameter %s from %s',
89             string, here)
90         items = here.xpath((
91             '.*//*[local-name()="parameter"]'
92             '[contains(concat(@dictRef, " "),'
93             ' ":%s ")/*[local-name()="scalar"]]' % string)
94         logging.debug('items: %s', items)
95         if not items:
96             logging.debug('trying "property" instead')
97             items = here.xpath((
98                 '.*//*[local-name()="property"]'
99                 '[contains(concat(@dictRef, " "),'
100                 ' ":%s ")/*[local-name()="scalar"]]' % string)
101             logging.debug('items: %s', items)
102         if not items:
103             logging.debug('trying "identifier" instead')
104             items = here.xpath((
105                 '.*//*[local-name()="identifier"]'
106                 '[contains(concat(@convention, " "),'
107                 ' ":%s ")][@value]' % string)
108             logging.debug('items: %s', items)
109         if items:
110             logging.debug('first item: %s', items[0].attrib)
111             values = list(set([i.text or i.attrib.get('value') for i in items]))
112             value = values[0] or default
113             logging.debug('found parameter %s: "%s"', string, value)

```

Figure 5.40 (cont.) : Continuation of figure 5.40.

```

114         if len(values) > 1:
115             logging.warn(
116                 'found %d parameters where 1 expected: %s',
117                 len(values), values)
118             break
119         else:
120             found = here.xpath('.//*')
121             logging.debug('found only: %s', found)
122             logging.debug('%s', [etree.tostring(f) for f in found
123                                 if f.tag.endswith('parameter')])
124             logging.debug('starting over at parent element')
125             here = here.getparent()
126     return value
127
128
129
130 def append_parameter_value(dictionary, element, field, default=''):
131     """
132     Append field:value to 'fields' key of dictionary
133
134     >>> tree = etree.fromstring(
135     ...     '<wrapper>\n'
136     ...     '<a><b><c><d><e>stuff</e></d></c></b></a>\n'
137     ...     '<module><module><parameter dictRef="cc:task">\n'
138     ...     '  <scalar dataType="xsd:string" id="copy.0">energy</scalar>\n'
139     ...     '</parameter></module></module>\n'
140     ...     '</wrapper>\n')
141     >>> element = tree.xpath('//e')[0]
142     >>> results = {'fields': {}}
143     >>> append_parameter_value(results, element, 'task')
144     >>> results
145     {'fields': {'task': 'energy'}}
146     """
147     logging.debug('append_parameter_value: element %s', element)
148     dictionary['fields'][snake(field)] = parameter(element, field, default)
149
150

```

Figure 5.40 (cont.) : Continuation of figure 5.40.


```

150
151 def snake(camelcased):
152     """
153     change programVersion to program_version, for database field ('snake case')
154
155     >>> snake('programVersion')
156     'program_version'
157
158     >>> snake('thisIsATest')
159     'this_is_a_test'
160     """
161     return re.sub('([A-Z])', r'_\1', camelcased).lower()
162
163 if __name__ == '__main__':
164     print(json.dumps(convert(sys.argv[1], sys.argv[2:])))
165

```

Figure 5.40 (cont.) : Continuation of figure 5.40.

5.15 Implementing Beauty

One of the requirements is to make the application visually appealing. This was done through the use of CSS3. A detailed explanation of what CSS does has been given in the Subsection 3.13.4. Django saves and serves static files and images through the use of the built-in static application. From the CSS file, we gave each of the pages specific colours. Also we used CSS to style the table and the appearance of the visualized molecule in both single molecule item view, multiple molecule item view. Figure 5.41 shows the CSS code used in styling our application.

```

1  body
2  {
3    background: #E0E0E0;
4  }
5
6  table, tr, td {border: 1px solid black;}
7
8  table {
9    border-collapse: collapse;
10   width: 100%;
11 }
12
13 th, td {
14   text-align: center;
15   padding: 15px;
16 }
17
18 tr:nth-child(even){background-color: #f2f2f2}
19
20 th {
21   background-color: #4CAF50;
22   color: white;
23 }
24
25
26 table, tr, td {border: 1px solid black;}
27 .container {height: 100%; width: 95%; padding: 1vw 1vw 0 1vw;}
28 .inline {float: left; height: 100%;}
29 span.labeled_input {white-space: nowrap;}
30 div.multiview .inline {margin-right: 5px;}
31 div.multiview .controls {margin: 10px 0 10px 0;}
32 div.singleview .controls {float: right;}
33 div.multiview .square {margin-bottom: 10px;}
34 div.singleview .square {float: left; margin-bottom: 10px;}
35 div.singleview div.molecule_data {clear: both; margin-top: 10px; width: 90%;}
36 div.multiview div.molecule_data {margin: 10px 0 10px 0;}
37 span.truncate {
38   width: auto;
39   white-space: nowrap;
40   overflow: hidden;
41   text-overflow: ellipsis;
42 }
43 div.header {text-align: left;}
44 .molecule_data, .controls {
45   padding: 5px;
46   border: 1px dotted black;
47 }

```

Figure 5.41: The application CSS style sheet.

Chapter 6

Tests and Verifications

6.1 Introduction

This chapter covers the various tests that were performed on the application, with the main focus on the functionality of the application.

6.2 Tests Outline

The main objective of this section is to make sure that the application meets all the defined specifications in Chapter 4. It seeks to resolve programming errors, defects, bugs, compatibility issues, and ensure adequate requirement coverage. We have defined our test outline and limited the scope to the under-listed below:

- Functional Testing.
- Compatibility Testing.
- Regression Testing.

6.3 Functional Testing

This test deals with all the required functionality of our web application. It seeks to ensure that the visualization application is tested and conforms with all the specifications. With Selenium, we were able to verify all the functionality through the written automated test cases, as a core component of TDD. Moreover, to further ensure absolute correctness, we conducted a series of manual tests as well. We verified all the functionality listed in Chapter 4 including the ones listed below:

- All outgoing links from pages were tested and verified.
- All buttons that trigger the model information (ball and sticks, ESP, transparent, VDW, SAS) works perfectly.
- Proper positioning of JSmol applet was verified.
- Extracted molecule information displayed on visualization was verified for correctness.
- Navigations were tested and verified.
- The views were tested (list, upload, item (single and multiple molecule)).
- Search box in list view work perfectly.
- Visualizations were tested and verified.

Each of the links goes to the right view, and all the buttons (radio and check-boxes) worked as expected by loading different models of the molecules. The extracted information displayed correctly.

6.3.1 Form Validation

As an important component of the functionality, we validated the forms. A web form allows one to enter data to be sent to a server for processing. We verified that each of the forms have the correct inputs. We tested and validated the HTML, CSS and JSmol codes for possible syntax errors.

6.3.2 Information Extraction Testing

A key component of the application is information extraction while handling uploading of large CML trees. This was tested and verified manually. The extracted information was displayed for each of the entries loaded and opened in either the single molecule item view or multiple molecule item view.

6.3.3 Visualization

As seen in the figures in Chapter 5, our application can visualize entries from the database in different models with their corresponding extracted stored parameters from the uploaded CML trees stored in the database.

6.4 Compatibility Test

This is a very important evaluation aspect of web applications. It involves testing browser compatibility as well as operating system compatibility. We tested the application on Windows, Mac OS and Ubuntu. We observed all the functionality, as well as how contents are rendered. The application was found to be compatible with Firefox, Internet explorer, Safari and Chrome. Furthermore, the application ran perfectly under Windows 10, Mac OS, and Ubuntu.

6.5 Regression Testing

Because of the newly added requirement including RESTful API implementation, we performed regression testing to ensure that our application still behaves according to specification. We did this by re-testing (i.e re-running all the test cases) in order to ensure that no error or defect has been introduced as a result of the changes in the requirements and the newly added secondary requirements during development. We also accessed stored entries directly from the database with their ID number from the URL and it worked perfectly.

Chapter 7

Conclusions

A test-centered software development approach has been examined for use in the development of a scientific web application for visualizing chemical structures based on the requirements in Chapter 4. TDD employs the use of automated tests to drive the design of a software with promises to produce high-quality applications with clean, maintainable code, easier documentation and easier integration of changing requirements. The web application was created using TDD with Django, Python3, JavaScript, HTML5, CSS3, JSmol, Selenium, and the RESTful API. Then, the scenarios were defined for testing and the application was tested according to them. The results were presented, and we verified that the implemented functionalities work.

TDD as a design protocol has no doubt set a strong foundation for adding additional functionality and integrating new or changing requirements in a simple manner. Going by the results of this project, TDD no doubt enhances application quality due to its test-centered nature. This is because every piece of functionality is written as a test and this gives no room for regression or logic bugs at the barest minimum. Furthermore, it incites to write simpler and testable code in smaller, easier-to-test chunks which invariably leads to better design. Simpler and more testable code in

most cases is faster and facilitates maintainability as it becomes easier to understand by other developers. TDD also makes code reusable, as it has been used in both testing and production environments. TDD ensures that all the required functionality are met.

However, we make no claim that TDD is a magic wand that solves all software problems. Rather, the approach has the prospect of ensuring high-quality applications if applied with all sense of discipline, as it can be a bit tedious especially in the beginning. The results of this research should be adopted in the software industry as it has added to the body of knowledge in this research area. More specifically, it could be extended to building high-quality requirement web applications in the academic circle and industry and thus raise the standards of developed applications.

7.1 Future Work

The future work would be to enhance the developed application to have a Graphical User Interface. Also, a more robust web server could be integrated as well as a better database. More visualization controls could be added as well. These improvements are valuable for the future versions of the application.

Bibliography

- [1] Scott Ambler. Test-driven development of relational databases. *IEEE Software*, 24(3):37–43, 2007.
- [2] Dave Astels. *Test driven development: A practical guide*. Prentice Hall Professional Technical Reference, 2003.
- [3] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.
- [4] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [5] Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 356–363. ACM, 2006.
- [6] Chris Cannam. Test driven development, 2011. URL <http://soundsoftware.ac.uk/unit-testing-why-bother/>. Accessed: 2017-01-11.
- [7] Chambers.com. Milestone definition, 2017. URL <http://www.chambers.com.au/glossary/milestone.php>. Accessed: 2017-02-20.
- [8] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew Ko. Let’s go to the

- whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 557–566. ACM, 2007.
- [9] Jim Conallen. Modeling web application architectures with UML. *Communication. ACM*, 42(10):63–70, 1999.
 - [10] Django. Design philosophies, 2015. URL <https://docs.djangoproject.com/en/1.10/misc/design-philosophies/>. Accessed: 2017-02-20.
 - [11] Django. How to provide initial data to Django using fixtures, 2017. URL <https://docs.djangoproject.com/en/1.10/howto/initial-data/>. Accessed: 2016-10-30.
 - [12] Django. Documentation on models, 2017. URL <https://docs.djangoproject.com/en/1.10/topics/db/models/>. Accessed: 2017-01-11.
 - [13] Django. Django Model View Template (MVT) documentation, 2017. URL <https://docs.djangoproject.com/en/dev/faq/general>. Accessed: 2017-02-25.
 - [14] Django. Documentation on the Staticfiles app, 2017. URL <https://docs.djangoproject.com/en/1.10/ref/contrib/staticfiles/>. Accessed: 2017-01-10.
 - [15] Django. Testing in Django, 2017. URL <https://docs.djangoproject.com/en/1.10/topics/testing/>. Accessed: 2017-02-10.
 - [16] Django. Django documentation on file upload, 2017. URL <https://docs.djangoproject.com/en/1.10/topics/http/file-uploads/>. Accessed: 2017-01-13.
 - [17] Selenium-Python Binding Documentation. Selenium Python binding, 2017. URL

- <http://selenium-python.readthedocs.io/locating-elements.html>. Accessed: 2016-09-25.
- [18] Tomaž Dogša and David Batič. The effectiveness of test-driven development: An industrial case study. *Software Quality Journal*, 19(4):643–661, 2011.
 - [19] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on software Engineering*, 31(3):226–237, 2005.
 - [20] Mozilla Foundation. Javascript, 2017. URL <https://developer.mozilla.org/en-US/docs/Web/JavaScript>. Accessed: 2017-04-01.
 - [21] Martin Fowler. Test driven development, 2005. URL <http://martinfowler.com/bliki/TestDrivenDevelopment.html>.
 - [22] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 2009.
 - [23] Bobby George and Laurie Williams. An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139. ACM, 2003.
 - [24] Atul Gupta and Pankaj Jalote. An experimental evaluation of the effectiveness and efficiency of the test driven development. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 285–294. IEEE, 2007.
 - [25] David Hansen. TDD is dead long live testing!, 2014. URL <http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html>. Accessed: 2017-01-11.
 - [26] Gemma Holliday, Peter Murray-Rust, and Henry Rzepa. Chemical markup,

- XML, and the world wide web. 6. CMLReact, an XML vocabulary for chemical reactions. *Journal of chemical information and modeling*, 46(1):145–157, 2006.
- [27] David Janzen and Hossein Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.
- [28] David Janzen and Hossein Saiedian. On the influence of test-driven development on software design. In *19th Conference on Software Engineering Education Training (CSEET'06)*, pages 141–148, 2006.
- [29] David Janzen and Hossein Saiedian. Does test-driven development really improve software design quality? *IEEE Software*, 25(2):77–84, 2008.
- [30] Mehdi Jazayeri. Some trends in web application development. In *2007 Future of Software Engineering*, pages 199–213. IEEE Computer Society, 2007.
- [31] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme programming installed*. Addison-Wesley Professional, 2001.
- [32] Dave Kuhlman. *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. Dave Kuhlman, 2009.
- [33] Stefan Kuhn, Tobias Helmus, Robert Lancashire, Peter Murray-Rust, Henry Rzepa, Christoph Steinbeck, and Egon Willighagen. Chemical markup, XML, and the world wide web. 7. CMLSpect, an XML vocabulary for spectral data. *Journal of chemical information and modeling*, 47(6):2015–2034, 2007.
- [34] Crispin Lisa. Driving software quality: How test-driven development impacts software quality. *IEEE Software*, 23(6):30–37, 2006.
- [35] Sept Mark. Rapid GUI programming with Python and Qt. *Learner's Guide to PyQt Programming*, 2, 2009.

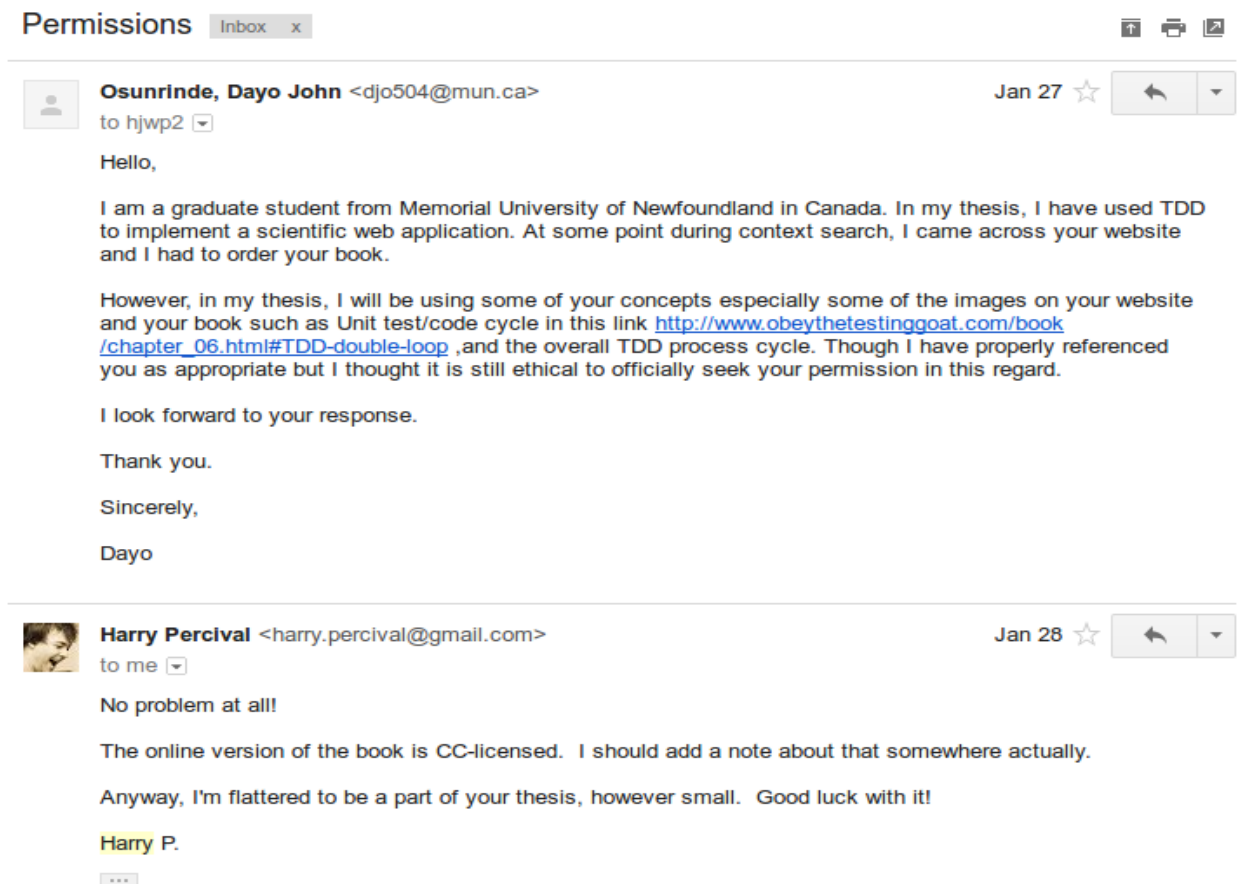
- [36] Bob Martin. Monogamous TDD: Uncle Bob’s response to David Hansen TDD is dead, 2014. URL <http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html>. Accessed: 2017-01-11.
- [37] Robert Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
- [38] Michael Maximilien and Laurie Williams. Assessing test-driven development at IBM. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, volume 10, pages 564–569. IEEE, 2003.
- [39] Steve McConnell. *Code complete*. Pearson Education, 2004.
- [40] Matthias Muller and Oliver Hagner. Experiment about test-first programming. *IEEE Proceedings-Software*, 149(5):131–136, 2002.
- [41] Peter Murray-Rust and Henry Rzepa. Chemical markup, XML, and the world-wide web. 1. basic principles. *Journal of Chemical Information and Computer Sciences*, 39(6):928–942, 1999.
- [42] Peter Murray-Rust and Henry S Rzepa. Chemical markup, XML, and the world wide web. 4. CML schema. *Journal of chemical information and computer sciences*, 43(3):757–772, 2003.
- [43] Peter Murray-Rust and Henry S Rzepa. CML: Evolution and design. *Journal of cheminformatics*, 3(1):44, 2011.
- [44] Ispir Mustafa. *Test Driven Development of Embedded Systems*. PhD thesis, Middle East Technical University, 2004.
- [45] Nachiappan Nagappan, Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing quality improvement through test driven development: re-

- sults and experiences of four industrial teams. *Empirical Software Engineering*, 13:289–302, 2008.
- [46] Harry Percival. *Test-driven development with Python*. " O'Reilly Media, Inc.", 2014.
 - [47] Pycharm. Pycharm IDE, 2017. URL <http://research.omicsgroup.org/index.php/PyCharm>. Accessed: 2017-03-30.
 - [48] ReviverSoft. .CML File Extension, 2017. URL <https://www.reviversoft.com/file-extensions/cml>. Accessed: 2017-06-19.
 - [49] Stephan Richter. lxml-XML and HTML with Python, 2017. URL <http://lxml.de/>. Accessed: 2017-05-11.
 - [50] SeleniumHQ. Selenium WebDriver documentation, 2017. URL <http://www.seleniumhq.org/projects/webdriver/>. Accessed: 2017-02-25.
 - [51] Zed A Shaw. Learn python the hard way, 2010.
 - [52] Technopedia. What is HTML5?, 2017. URL www.techopedia.com/definition/1892/hypertext-markup-language-html. Accessed: 2017-02-25.
 - [53] Technopedia. Web programming definition, 2017. URL www.techopedia.com/definition/23898/web-programming. Accessed: 2017-02-23.
 - [54] Tutorialspoint.com. RESTful web services introduction, 2017. URL https://www.tutorialspoint.com/restful/restful_introduction.htm. Accessed: 2017-01-30.
 - [55] Jagoda Walny, Jonathan Haber, Marian Dörk, Jonathan Sillito, and Sheelagh Carpendale. Follow that sketch: Lifecycles of diagrams and sketches in software development. In *Visualizing Software for Understanding and Analysis (VIS-SOFT)*, 2011 6th IEEE International Workshop on, pages 1–8. IEEE, 2011.

- [56] Hans Wasmus, Hans-Gerhard Gross, Cesar Gonzales-Perez, and Maciaszek leszek. Evaluation of test-driven development. In *2nd Working Conference on Evaluation of Novel Approaches to Software Engineering*, volume 30, pages 103–110. Insticc Press, 2007.
- [57] Learn Python website. Python functions. URL <http://www.learnpython.org/>.
- [58] W3 Website. Cascading Style Sheets (CSS), 2017. URL <https://www.w3.org/standards/webdesign/htmlcss>. Accessed: 2017-03-02.
- [59] W3 Website. Hypertext Markup Language (HTML), 2017. URL <https://www.w3.org/standards/webdesign/htmlcss>. Accessed: 2017-02-25.
- [60] Wiki. JSmol documentation, 2017. URL http://wiki.jmol.org/index.php/Jmol_JavaScript_Object#JSmol. Accessed: 2017-02-25.
- [61] Wiki. Python documentation, 2017. URL <https://wiki.python.org/moin/BeginnersGuide/Overview>. Accessed: 2017-02-25.

Appendix A

Email Permission*



*This appendix provides the permission email for Overall TDD Process and TDD with functional and unit test(Figure 2.3 and Figure 2.4)